CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Transportability Guideline for Ada Real-Time Software**

CIN: **C02 092LA 0008**

31 MAY 1989

TRANSPORTABILITY GUIDELINE FOR

ADA REAL-TIME SOFTWARE

PREPARED BY:

LABTEK CORPORATION
8 LUNAR DRIVE
WOODBRIDGE, CT 06525

SPONSORING ORGANIZATION:

U.S. ARMY HQ CECOM
CENTER FOR SOFTWARE ENGINEERING

DATE:

24 APRIL 1989

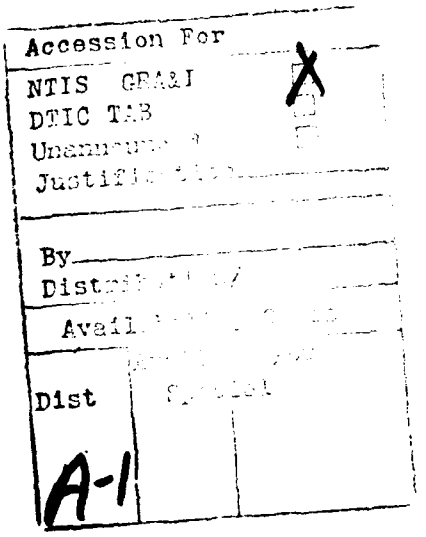
CONTRACT NO. DAAL03-86-D-0001

DELIVERY ORDER NUMBER: 0731
SCIENTIFIC SERVICES PROGRAM

Table of Contents

Table of Contents

List of Figures



1. Introduction

This document reports the findings of a project that analyzed issues involved with the transporting of real-time Ada programs.

1.1 Background

Software transportability is one of the cost-saving benefits anticipated with the use of Ada. Support for transportability was a major goal in the design of the Ada language. However, transportability is not automatic with the use of Ada; programs written without specific attention to transportability will not, in general, be transportable.

Software must be transportable in order to take advantage of rapid changes in both processor technology and compiler technology. The time between processor generations is typically less than three years, whereas the application software must endure for a period of 10-20 years. Current compilers are evolving rapidly. Performance gains due to optimization are eagerly awaited in each new compiler release.

For real-time embedded applications, software test-on-host and integration-on-target requires that the application program be transportable to at least two different computers (typically). Programming teams must develop software on a host environment, perform as much testing as possible on the host, and then transport the code to the target environment for further testing and integration. Again, designing software to be transportable will be an aid in this process.

Software transportability is also a prerequisite for a "Software First" methodology. This methodology purports designing the application software without regard for the underlying hardware. In the past, hardware selection was the driving cost associated with a system. Today, the *custom* application software is the driving cost. Much of the hardware can be bought off-the-shelf.

Transportability is not an absolute, but rather a measure of degree. It is unusual to obtain 100% transportable software when working with real-time embedded applications. Rather, the goal is to maximize the transportability in the face of other programming constraints. Tradeoffs are frequently required to achieve the appropriate degree of transportability while obtaining sufficient performance and program clarity.

Transportability does not come without additional costs. Initial training of the staff must be included in the budgeting for resources. Designing for transportability is usually more difficult because the design is further constrained by the need to limit implementation dependencies. The coding of the software is also hindered because the programmer must always be concerned that each of the guidelines is met. This often requires writing more complicated code to achieve the same effect. Coordination among the programmers is also more important to maximize the commonality within a program.

1.2 Purpose and Intent

The purpose of this report is to develop an initial set of guidelines for writing transportable Ada programs for embedded real-time applications. Three transportability guideline reports

have already been written. The first, in 1982, titled "Ada-Europe Guidelines for the Portability of Ada Programs", by Nissan, Wallis, Wichmann and others [2], and the second in 1985, titled "Ada Portability Guidelines", by SofTech, Inc., [3]. The third report was just recently published in February 1988 by Software Productivity Consortium, titled "Ada Style Guide". [11] Work in this area is also currently underway at the Software Engineering Institute. The intent of this report is not to duplicate previous work, but rather, to produce a guideline which focuses on dealing with the implementation dependencies allowed within the Ada language to achieve transportable software for real-time applications.

The Catalogue of Ada Runtime Implementation Dependencies (CRID), initially produced by the Ada Runtime Environment Working Group (ARTEWG) of SIGAda, and subsequently enhanced by the Center for Software Engineering (CSE), Fort Monmouth, NJ [9], is an important input into this work. It details the areas of the Ada language where the language definition has left the implementation details up to the Ada compiler writer. These are the areas of the language which will not necessarily transport. This guide will show how to handle these implementation dependencies.

1.3 *Intended Audience*

The intended audience of this guide includes those individuals trying to port software, evaluate software for transportability, or perform Ada design, code, and implementation. This guide may also be beneficial to software project management personnel.

1.4 Organization of Document

Section one contains background information as well as the purpose and intent of this work.

Section two details the approach used to gather the information and the criteria used for its evaluation.

Section three is a technical discussion of the following:

- Definition of transportability and its relationship to reusability for real-time embedded applications.
- How the Ada language enhances and/or impedes the process of producing transportable real-time programs.

Section four discusses the interrelationship of transportability with real-time programming. It contains representative benchmarks which demonstrate how to evaluate the relative performance of transportable software compared with more implementation specific software.

Section five assimilates all of the information into a guide for writing transportable Ada software for real-time embedded applications. These guidelines are in the form of recommendations on how to use the real-time features of Ada.

Section six contains a summary of the results.

Section seven contains the reference materials used in the creation of this report.

Section eight contains a glossary of terms used in this report.

2. Approach

The approach used to obtain the information in this report was:

1. Review the current literature, especially the ARTEWG and CSE documents [8,9], for material relevant to this task.

2. Determine the relationship between transportability and reusability in real-time embedded applications.

3. Demonstrate how the Ada language enhances or impedes the process of producing transportable real-time programs.

4. Develop and execute representative benchmarks to test the performance of transportable software using two Ada compilers hosted and/or targeted for different machines and incorporate the results into the transportability guideline.

5. Show how transportability relates to performance in real-time embedded systems.

6. Analyze the input material obtained by steps 1-5 above and produce the guideline.

3. Technical Discussion

3.1 Definitions

3.1.1 Transportability

An Ada RunTime Environment (RTE) consists of three functional areas: abstract data structures, code sequences, and predefined subroutines. It includes all of the runtime support routines, the conventions between the runtime routines and the compiler, and the underlying virtual machine of the target computer. "Virtual" is used in the sense that it may be a machine with layered software (a host operating system). An RTE does not include the application itself, but includes everything the application can interact with. Each layer has a protocol between it and the layer underneath it for interfacing. In the event that there isn't any operating system layer (the bare- machine target), the runtime includes those low-level functions found in an operating system. [8] See Figure 1.

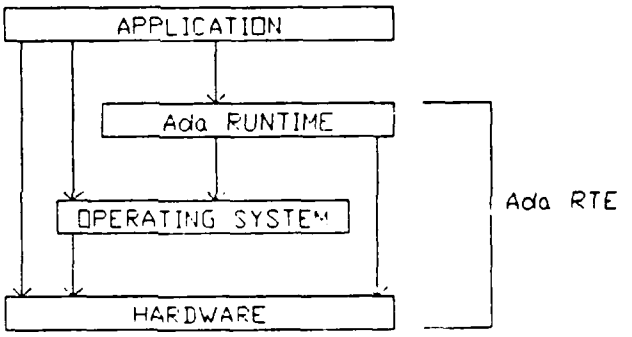


Figure 1. Ada Runtime Environment (RTE)

The RunTime System (RTS) is the set of subprograms, which may be invoked by linking, loading, and executing object code generated by an Ada compiler. If these subprograms use or depend upon the services of an operating system, then the target runtime system includes those portions of that operating system. [7] These predefined subroutines are chosen from the Runtime Library (RTL) for that Ada compilation system.

For this report, "transporting software" means to change the runtime environment (RTE) of an application. A change could be as small as using a different compiler or linker control, or much larger such as moving the application to a completely new target architecture and switching to a new compiler.

Transportability, then, is the measure of effort required to transport application software to a different runtime environment (RTE). For a measure, if it requires 1-person month to

transport software that required 20 person months to develop, the software is said to be 95% "transportable".

The shaded area in Figure 2. depicts the area of change when an application is transported. The application does not change (the larger circle representing the Application is not shaded), but the runtime environment does change (by definition, note the smaller shaded circle of runtime environment 2). Portions of the application code which directly interfaces to the runtime environment (note the shaded box around runtime environment 2) may also change.

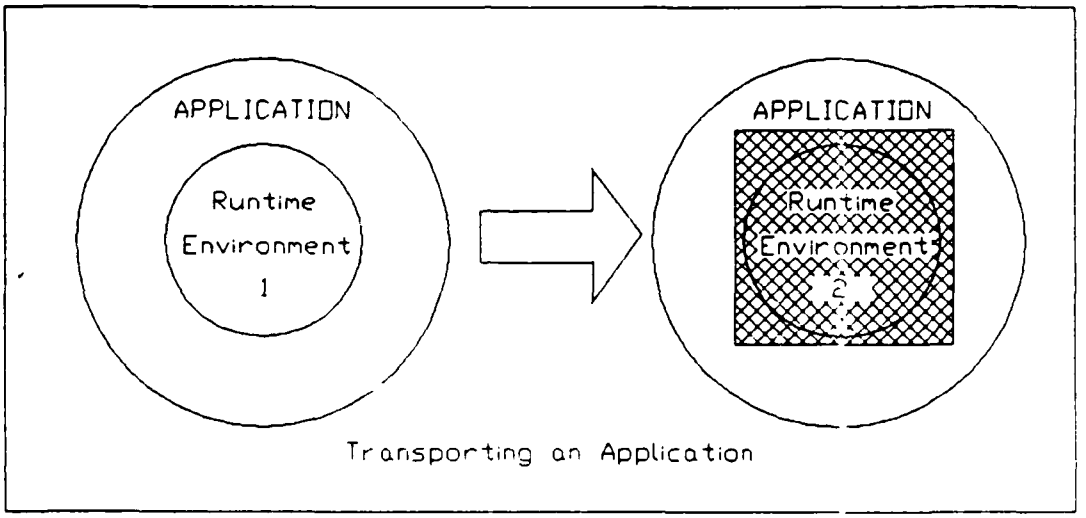


Figure 2. Transporting an Application

There are several common reasons for software transport:

a.) A new CPU. The next processor generation could become available and in order to take advantage of the increased processor performance it may be desirable to replace the existing CPU with it. In some cases, it may be necessary to change processor families altogether.

b.) A new compiler. A new version of the compiler may be released. In order to take advantage of its new features or to keep current it may be desirable to use this new compiler version.

c.) A new runtime library. The compiler vendor could supply a new version of the runtime library. This will produce a change in the RTE.

d.) Different compiler switches are enabled (optimization). The same code can work differently with different compiler switches enabled/disabled. A typical case in point is the optimization switch. Care must be taken to insure that the application code operates as desired after this switch is enabled.

e.) A change in underlying operating system. If the underlying operating system changes, this is essentially a change in the RTE. It is important to know what effect this has on the application software.

In developing transportable software, the primary objective is to reduce the difficulty in identifying and changing the parts of the program necessary to get acceptable program behavior on the new target. Special attention is required to insure that the new program does indeed perform with correct characteristics. An important goal is to force any required implementation dependencies that are different between the two targets to be identified during program compilation rather than during program execution.

3.1.2 Reusability

Reusability is a measure of effort required to use a software component in a new application. In order to be effectively reused, the component may have to be adapted to the requirements of the new application. [4]

The shaded area in Figure 3. below depicts the area of change when a component from Application 1 is reused in Application 2. The application itself changes (note the shading of the large circle for Application 2), but most of the reusable component is left intact (note the unshaded box within the smaller circle of Application 2). There may be portions of the reusable component which need modification (note the shaded portion of the smaller circle of Application 2) to operate correctly in the new application.

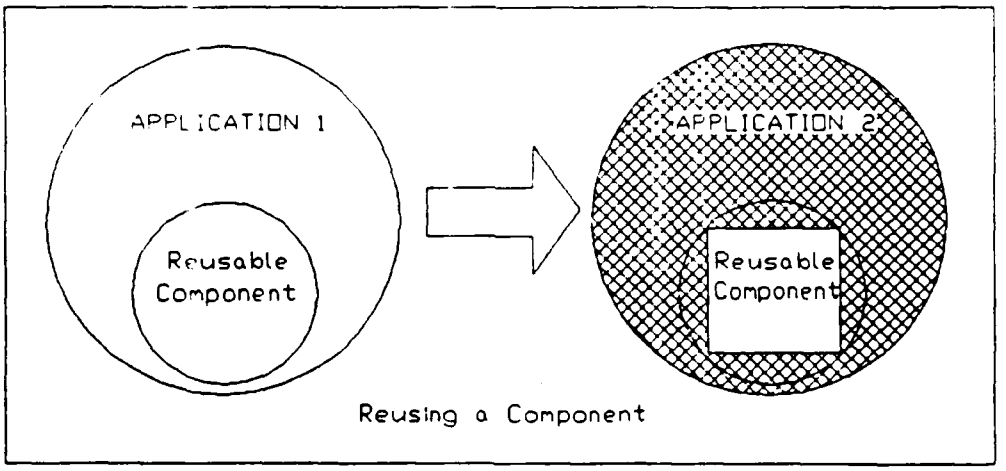


Figure 3. Reusing a Component

3.1.3 Distinction between Transportability and Reusability

The distinction between transportability and reusability is the following: Transportability is concerned with changes in the runtime environment, whereas reusability is concerned with

changes in application. It is possible to have code that is reused, transported, or both. Often transportability is a requirement of reuse, but not always.

There are several things to notice in the definition of transportability that make it distinct from reusability, as noted in [4].

- Transportability is normally concerned with transporting an entire application, whereas reusability is concerned with the reuse of a component of an application.

- When an application is transported, it is used in a new target environment; when a component is reused, it is used in a new application.

- Reusability is concerned with dealing with a different application which uses a component and any aspects of that component that need to change to reflect the requirements of the new application.

- To a large extent, reusability is a design consideration while transportability is an implementation consideration. Reusability is achieved primarily by control of the structure of the overall system and of the nature of the interfaces between components. Transportability is concerned more with specific use of language features so as to avoid undesired hardware (or other target environment) dependencies. A useful (although somewhat simplified) way to look at this is that reusability deals with software and interface dependencies, while transportability deals with hardware and system software dependencies.

Transportability of an application does not imply that the components of that application are reusable in another application. It is very likely that all components of the application could be tightly coupled, thus preventing any one of them from being used separately in another application. In general, all components of the application will be used in all target environments. There may be specific cases where application specific hardware interfaces may change when transporting occurs. In this case, the software component that interfaces with the hardware will also change.

Similarly, reusability does not imply transportability. A reusable component could be very target dependent, e.g., an I/O package that is reusable across all projects on one target environment, but unusable in any other target environment.

It is important to note, however, that transportability and reusability are not mutually exclusive properties. If it is desired to be able to reuse a component in application systems that run on different target environments, then the component must be designed for transportability as well as reusability. Thus, a goal of maximizing reusability will usually include transportability as a requirement.

3.2 Ada and Transportability

The Ada language does support many concepts which aid transportability. Among these concepts are: abstraction, encapsulation, and information hiding. Abstraction and encapsulation are supported by the package concept. Related subprograms can be grouped together and seen by a higher level as a single entity. Information hiding is enforced via strong typing, and the separation of package and subprogram specifications from their

respective bodies. Use of Predefined Language Attributes, found in Annex A of the RM [1] also aid transportability.

On the other hand, the Ada language was designed to be implementation independent. The designers of the language chose to do this to avoid tying it to current technology, so that advances in technology could be readily accepted. Consequently, there are many places in the language definition where implementers of the language are free to decide how a language feature is to be performed (as long as the feature conforms to the rules of the language). For example, the implementer is free to choose the mechanism of parameter passing for composite types. This choice, and all the other choices the implementer makes, may have both positive and negative effects on an application program especially in terms of its performance and its transportability. If the application has stringent requirements for either performance or transportability, then knowledge about the choices made in the various implementations will be useful.

The areas where the language definition has left the implementation details up to the Ada compiler writers are called *implementation dependencies*. These are the areas to be concerned about when writing software that is to be transportable.

4. Transportability Considerations for Real-Time Systems

This section discusses the interrelationship of transportability with real-time programming. Principal objectives for real-time programs are to keep them simple and fast. These two objectives are sometimes in conflict with the objective of transportability.

To enhance the transportability of embedded software, some techniques can be used to reduce the dependence on the compiler implementation. In some cases, these changes will actually improve performance at the expense of much more complex code. In other cases, some performance degradation can be expected. Below are the areas identified where the design can be modified to improve transportability. They are divided into two categories: complexity tradeoffs and performance tradeoffs.

4.1 Transportability vs. Complexity

Runtime routines to support access type allocators and manage the storage for collections vary significantly among implementations. By limiting the execution of allocators to program initialization and defining application specific routines to allocate and deallocate storage, a program can control the characteristics of the dynamic storage. This approach is useful if the purpose of dynamic allocation in the application is primarily to manage storage rather than trying to conserve on memory. The use of fixed size queues to manage the storage will generally provide better performance and transportability, although it may increase the complexity of the application software. A hybrid approach may be used to use Ada allocators to initially obtain the storage accessible via an array with elements of an access type. After the initial allocation, the storage is managed as needed via explicit calls to ALLOCATE and DEALLOCATE procedures defined in an application package. This can eliminate any dependence on the ability of the runtime to efficiently allocate and reclaim storage. This technique is not convenient when storage is shared among many different data types since it will require unchecked type conversions of the access types. Also unconstrained types are impractical to manage this way. Fortunately, there are many cases where fixed size buffers need to be made available for some unspecified period and then returned to a buffer pool. This type of application is fairly easy to implement and will provide predictable behavior.

Another example of adding complexity is with file management routines. Temporary files (as defined by Ada) need not be deleted in the same way by all implementations. Therefore, rather than using temporary files, a specific mechanism to create a unique filename and explicitly create and delete is preferable. This is unlikely to change the execution time significantly but does require more programming effort. Problems may arise are in creating a unique file name (that is suitable for many file systems), and making sure the application defined "temporary" files are deleted, even when the program is terminated by an exception.

4.2 Transportability vs. Performance

Applications that process external files will have difficulty in transporting anything other than strictly ASCII representations of data. By performing all input/output as ASCII text strings, dependence on the binary representation of the types is removed. For enumeration types, use 'POS/'VAL and transfer them as ASCII numbers. The following benchmark measures the relative performance of writing integers as binary values as opposed to a more transportable representation of ASCII numbers. The results of this benchmark indicated a

2:1 performance degradation due to using ASCII values rather than binary. Note also that the file size was a factor of 3:1 for ASCII text compared to binary.

```
--
--
--  Transportability Performance Test
--
--  This test is designed to measure relative performance of two different
--  representations of integer data for the purposes of transportability.
--
--  TEST:  Representation of Integer Data
--
--  The program writes out 10,000 values of an INTEGER type to disk and
--  the writes out the same data as TEXT so that it is more transportable.
--  Between each action the clock is read and the interval is displayed.
--
--

with Text_IO;
with Sequential_IO;
with Calendar;

procedure Trans_IO is
  package BINARY_IO is new Sequential_IO(INTEGER);
  BINARY_FILE     : BINARY_IO.FILE_TYPE;

  package Int_IO is new Text_IO.Integer_IO(INTEGER);
  INTEGER_FILE    : Text_IO.FILE_TYPE;
  INTEGER_VALUE   : INTEGER := 1234;

  START           : Calendar.TIME;
  STOP            : Calendar.TIME;

  BINARY_TIME     : DURATION;
  TEXT_TIME       : DURATION;

  package Duration_IO is new Text_IO.Fixed_IO(DURATION);

begin
  BINARY_IO.Create(BINARY_FILE,BINARY_IO.OUT_FILE,"BINARY.DAT");
  Text_IO.Put_Line("Writing binary data:");
  START := Calendar.Clock;
  for I in 1..10000 loop
    BINARY_IO.Write(BINARY_FILE,INTEGER_VALUE);
  end loop;
  STOP := Calendar.Clock;
  Text_IO.Put_Line("Binary Data Written");
  BINARY_IO.Close(BINARY_FILE);
  BINARY_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time to Write BINARY data: ");
  Duration_IO.Put(BINARY_TIME); Text_IO.New_Line;
```

```
  Text_IO.Create(INTEGER_FILE,Text_IO.OUT_FILE,"INT.DAT");
  Text_IO.Put_Line("Writing text data:");
  START := Calendar.Clock;
  for I in 1..10000 loop
    Int_IO.Put(INTEGER_FILE,INTEGER_VALUE);
  end loop;
  STOP := Calendar.Clock;
  Text_IO.Put_Line("Text Data Written");
  Text_IO.Close(INTEGER_FILE);
  TEXT_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time to Write TEXT   data: ");
  Duration_IO.Put(TEXT_TIME); Text_IO.New_Line;

end TRANS_IO;
```

Address clauses are not supported in a standard way among implementations. Typically, what is desired is a linear physical address that will access hardware on the system bus. To provide address clauses that are transportable, a function to translate a "uniform" addressing format (32-bit linear physical address) to a implementation dependent System.ADDRESS can be used. This will cause all address clauses to be nonstatic and therefore require that indirect references be made to the object. The following program can be used to help assess the overhead for references to objects located by nonstatic address clauses. The following benchmark tests the performance penalty for using nonstatic address clauses. One implementation failed to compile the benchmark (although static address clauses were supported). The vendor indicated that this problem would be fixed in the next release. Another implementation measured negligible performance degradation due to nonstatic address clauses. By examining the generated code, it was determined that this implementation always referenced objects specified by address clauses indirectly.

```
--
--  Transportability Performance Test
--
--  This test is designed to provide insight into what relative performance
--  degradation is likely to support translated addresses clauses (forcing
--  them to be nonstatic).
--
--  TEST: Address Clause Representation
--
--  The expression in address clauses is implementation dependent, yet
--  the primary reason for address clauses is to map program references
--  to specific HARDWARE addresses which can almost universally be expressed
--  as a linear 32-bit address (or some subset thereof).  To enhance
--  transportability, all address clauses can be expressed as a function call
--  which accepts a 32-bit hardware address and returns the appropriate
--  implementation defined representation System.ADDRESS.
--
--

with System;
```

```
package Transport_Functions is
  function Address (ADDR : LONG_INTEGER) return System.ADDRESS;
end Transport_Functions;

package body Transport_Functions is
  function Address (ADDR : LONG_INTEGER) return System.ADDRESS is
  begin
  -- System.ADDRESS is derived from LONG_INTEGER
    return System.ADDRESS(ADDR);
  end Address;

end Transport_Functions;


with Text_IO;
with Calendar;
with System;
with Transport_Functions;

procedure Tráns_ADDR is

  START           : Calendar.TIME;
  STOP            : Calendar.TIME;

  STATIC_TIME     : DURATION;
  NON_STATIC_TIME : DURATION;

  package Duration_IO is new Text_IO.Fixed_IO(DURATION);

-- first use implementation dependent approach
    ADDR1       : INTEGER;
      for ADDR1 use at 16#DC000#;   -- static value

-- then use transportable approach
    ADDR2       : INTEGER;
      for ADDR2 use at Transport_Functions.Address(16#DC000#); -- nonstatic

    TEMPORARY   : INTEGER;

begin
--
--  Use at least 20 references to reduce time attributed to loop construct
--
  START := Calendar.Clock;
  for I in 1..10000 loop
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;   -- 2
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;   -- 4
    TEMPORARY := ADDR1;
```

```
    TEMPORARY := ADDR1;    -- 6
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 8
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 10
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 12
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 14
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 16
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 18
    TEMPORARY := ADDR1;
    TEMPORARY := ADDR1;    -- 20
  end loop;
  STOP := Calendar.Clock;
  STATIC_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time to Read Object with Static Address Clause: ");
  Duration_IO,Put(STATIC_TIME); Text_IO.New_Line;

  START := Calendar.Clock;
  for I in 1..10000 loop
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 2
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 4
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 6
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 8
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 10
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 12
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 14
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 16
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 18
    TEMPORARY := ADDR2;
    TEMPORARY := ADDR2;    -- 20
  end loop;
  STOP := Calendar.Clock;
  NON_STATIC_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time to Read Object with Nonstatic Address Clause: ");
  Duration_IO.Put(NON_STATIC_TIME); Text_IO.New_Line;

end TRANS_ADDR;
```

Exception propagation time is extremely implementation dependent. To avoid using exceptions for errors that can be expected, if statements can be used to check for errors that might otherwise be processed by exception handlers. This will give more consistent timing to program execution since dependence on exception propagation is eliminated. The following benchmark assumes some data is being obtained from a hardware device. Although the range is acceptable on input, occasionally an offset has to be added. If the hardware is working properly, this addition would never result in an exception. Rather than depending on the exception and the associated overhead, the error condition is explicitly tested. The additional overhead for the check necessary to detect the error and the check to catch propagation of the error are being measured. The cost of this approach is highly dependent on how many checks must be made, and the size and nesting level of procedures which perform the checks. Also, if the error flag(s) are passed as parameters rather than globally accessed, additional time would be consumed. Note that if exceptions are only used for serious failures and the response time from such an exception is not critical, the normal Ada exception mechanism is preferred over the "flag" approach.

```
--
--  Transportability Performance Test
--
--  This test is designed to provide insight into what relative performance
--  degradation is likely to support provide manual checking of subprogram
--  result conditions as opposed to using the built in Ada exception
--  mechanism.
--
--  TEST: Exceptions for standard error condition reporting
--
--  This test is designed to manually test a flag as opposed to using
--  exceptions to get more consistent error handling performance when
--  transported.
--
--

with Text_IO;
with Calendar;

procedure Trans_Except is

--
--  TIMING DECLARATIONS
--
  START           : Calendar.TIME;
  STOP            : Calendar.TIME;
  EXCEPT_TIME     : DURATION;
  FLAG_TIME       : DURATION;

  package Duration_IO is new Text_IO.Fixed_IO(DURATION);

-----------------------------

  ERROR_CONDITION : BOOLEAN := FALSE;
```

```
  type TEMPERATURE_TYPE is range 0..1000;

  NORMAL_VALUE :  TEMPERATURE_TYPE := 10;
  RESULT       :  TEMPERATURE_TYPE;
  limit        : constant TEMPERATURE_TYPE := 800;
  OFFSET_TEMP  : constant TEMPERATURE_TYPE := 200;

--
--  sample procedure would normally process an incoming temperature
--  from a hardware device, which can typically range from 0 to 1000
--  degrees, but this procedure requires the temperature to be 0 to 800
--  degrees.
--
  procedure DO_SOMETHING(UNKNOWN_INPUT : TEMPERATURE_TYPE) is
  begin
    if UNKNOWN_INPUT > limit then
      ERROR_CONDITION := TRUE;
      return;
    else
      RESULT := UNKNOWN_INPUT + OFFSET_TEMP;
    end if;
  end DO_SOMETHING;

  procedure DO_SOMETHING2(UNKNOWN_INPUT : TEMPERATURE_TYPE) is
  begin
    RESULT := UNKNOWN_INPUT + OFFSET_TEMP;  -- simply do it, let exception occur
  end DO_SOMETHING2;

begin
  Text_IO.Put_Line("Running Exception Benchmark");
  START := Calendar.Clock;
  for I in 1..10_000_000 loop
    DO_SOMETHING(NORMAL_VALUE);
    exit when ERROR_CONDITION;     -- any check for exception in procedure
  end loop;
  STOP := Calendar.Clock;
  FLAG_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time for transportable exception handling: ");
  Duration_IO.Put(FLAG_TIME); Text_IO.New_Line;

  START := Calendar.Clock;
  for I in 1..10_000_000 loop
    DO_SOMETHING(NORMAL_VALUE);    -- no check necessary, depend on exception
  end loop;
  STOP := Calendar.Clock;
  EXCEPT_TIME := Calendar."-"(STOP,START);
  Text_IO.Put("Time for normal exception technique: ");
  Duration_IO.Put(EXCEPT_TIME); Text_IO.New_Line;

end Trans_Except;
```

5. Transportability Guideline

The following guidelines were produced from studying the implementation dependencies allowed within the Ada language and from utilizing work performed prior to this effort ([2], [3], [11]). The Catalogue of Ada Runtime Implementation Dependencies [9] details the areas of the Ada language where the Ada implementers could make design decisions as to how a particular language feature was implemented. The catalogue should be consulted as a further reference to this work.

It is important to reiterate the fact that this guideline is concerned with handling the implementation dependencies allowed within the Ada language to achieve transportable real-time embedded software. Previous guides provided useful transportability guidelines for the language constructs, in general. It is not our intent to duplicate this work, although there will be some overlap. The reader is referred to these reports for additional information if needed. [2] [3] [11]

Previous guidelines typically attempted to completely restrict programs from using any implementation dependent features of Ada. Because of the large number of implementation dependencies, this "Greatest Common Divisor" approach is inadequate for real-time embedded applications. This guideline instead insists that the implementation requirements of the application are clearly specified in the source code. For example, if a calculation requires a range of 1 to 70,000, this range must be specified for the types of those objects. In this way, no assumptions are made with respect to capacities of the target (in the example, the range of type "INTEGER"). Rather, they are verified by the compiler for each target. This restricts transportability to compilation systems that fully support the source code specifications. In practice this does not reduce the number of useful implementations, since it is unlikely the real-time system would operate if the specified capability was not properly supported and had to be constructed from other primitives.

The implementation dependencies were analyzed to determine which dependencies are most likely to create a transportability problem. These selected dependencies were further analyzed to determine how programs could be specified that would limit the impact of their use. Where appropriate, benchmarks were written to determine what impact the more transportable approach had on execution performance. Cases where the language does not permit a translation were identified, and for these cases the most common implementation approach was determined using vendor documentation and benchmarks (see Appendix A). This information was synthesized and used to produce the following list of guidelines.

The general strategy is to improve the transportability of real-time programs by addressing five major areas of concern:

1) Eliminate the need for and use of Implementation Dependencies wherever practical.

2) When dependencies are required to meet system requirements, provide a translation from a more universal approach to the implementation specific approach wherever possible. The translation function(s) would then require modification for each transport.

3) When translations are not possible, use the simplest and most conventional approach available. Document these so that the work of transporting the software is clearly understood. Pay special attention to dependencies that could be interpreted incorrectly (and without any warning) by some compilation systems.

4) As always, a clear design and good documentation provide the best assurance that a program can be maintained, including transporting to new runtime environments. If ease of transportability is especially critical, the program should be targeted to two different processor architectures concurrently during the initial development effort. This will help to provide identification of problems that restrict transportability in time to correct them. All documentation related to transporting should be collected in a document titled "Transporting Manual". This manual serves as a collection point for implementation dependent requirements that can not be verified by the compiler. Areas identified in this manual should be given priority during the transporting effort, as they are likely to be the most serious to resolve. Specifically including in the manual is the "Transportability Requirements Report" (see Appendix B) which summarizes the major implementation dependencies of the application.

5) When possible, application specific hardware that interacts directly with the software should be moved to the new target system along with the software. If this is not feasible, the transported software will have to be modified to work with the new application specific hardware. To facilitate this, modularize all application specific hardware references so that it can be replaced easily and dependencies on it can be identified.

Before attempting to transport software, there are minimum requirements for both the original target compilation system and the destination target compilation system. Transporting software in their absence is extremely risky from both reliability and cost/schedule perspectives. These include:

Pragmas that do not achieve their desired functions must generate a warning message. This includes **pragma** "PACK", where full bit packing density is not achieved.

All representation clauses should be supported to the extent that is reasonable for the target processor.

A reliable approach to determining task stack size must be supported.

Also, a method to obtain intermixed listings of Ada and machine code is frequently a necessity, especially if reliability is a concern.

Specific recommendations are divided into categories to which they are most related.

5.1 Erroneous Programs and Incorrect Order Dependencies

An erroneous program is a program that is incorrect, but detection is not required by an Ada compiler. In some cases execution will raise "PROGRAM_ERROR".

Guide(01): Programmers should be aware of what mechanisms produce erroneous programs. Care must be taken to avoid these mechanisms. Checks for those constructs should be included in code walk-throughs. The checklist should include:

- Reference to uninitialized variables
- Unsynchronized access to shared data
- Access of deallocated objects
- Invalid unchecked conversions
- Invalid change of a discriminate value
- Dependence on parameter passing mechanism
- Multiple address clauses for overlaid entities
- Invalid suppression of exception check
- Possibility of all **accept** alternatives closed in a selective wait without **else**
- Reaching end of function body (without return)

An incorrect order dependency is a specification for some constructs that are to be executed in some order that is not defined by the language. This means the implementation is allowed to execute these parts in any given order. Therefore, different implementations can produce different results.

Guide(02): Incorrect order dependencies should not exist in well designed programs. The following is a checklist where incorrect order dependencies can occur. Check for:

- Evaluation of default expressions - RM 3.2.1(15)
- Range constraint evaluation - RM 3.5(5)
- Index evaluation order - RM 3.6(10)
- Component subtype elaboration order - RM 3.6(10)
- Index constraints - RM 3.6.1(11)
- Discriminate checks for incomplete types - RM 3.7.2(5)
- Discriminate evaluation order - RM 3.7.2(13)
- Elaboration checks and parameter evaluation - RM 3.9(5)
- Evaluation of an indexed component - RM 4.1.1(4)
- Evaluation of a slice - RM 4.1.2(4)
- Evaluation of the component expressions of a record aggregate - RM 4.3.1(3)
- Evaluation order of component associations - RM 4.3.2(10)
- Order of constraint checking - RM 4.3.2(11)
- Evaluation of operands in an expression - RM 4.5(5)
- Assignment statement evaluation - RM 5.2(3).1, 5.2(4)
- Order of evaluation of parameter associations - RM 6.4(6)
- Order of parameter copy-back - RM 6.4(6)
- Task activation order - RM 9.3(1)
- Guard condition evaluation - RM 9.7.1(5)
- Evaluation of **delay** expression or entry family index - RM 9.7.1(5)
- Selective wait alternatives - RM 9.7.1(6)
- Scheduling order of tasks - RM 9.8(5)
- Order of abortion - RM 9.10(4)
- Elaboration order of compilation units - RM 10.5(2)
- Elaboration of generic instantiations - RM 12.3(17)

5.2 Storage Issues

Guide(03): If memory space is limited, determine how specific the linker (binder) is when selecting data and code for inclusion into the executable image. This often is different for runtime support routines and application routines. Some implementations load the entire **package** even if only one data object is referenced. Others select only what is referenced.

Guide(04): For array types which must have exact storage layout requirements, use a length clause for the entire object and insure that the number of elements multiplied by the bits specified (or required) for all possible values of the element type is exactly equal to the number of bits specified in the length clause. If a record is specified as the element type, use a record representation clause to completely specify the layout.

Example:

```
type COLOR_TYPE is (RED,BLUE,YELLOW,ORANGE);
for COLOR_TYPE'size use 2;

type COLOR_ARRAY_TYPE is array(1..5) of COLOR_T PE;
for COLOR_ARRAY_TYPE'size use 10;
```

This forces the upper bound to be 10, and since 5 (five) 2-bit elements are required to store the necessary information, the lower bound is also 10 bits. Note: **pragma** "PACK" is the preferred approach to obtaining the desired bit density for arrays (see Guide 7 and minimum requirements listed above).

Guide(05): If access types are used, verify that sufficient space is made available for each access collection. This should be done using a length clause on the "STORAGE_SIZE" attribute.

Guide(06): If tasking is used, verify that sufficient space is made available for task activation. This should be done using a length clause on the "STORAGE_SIZE" attribute. Always explicitly state the storage requirement for each task. This implies that all tasks are defined as task types. It is best if all storage requirements are specified in terms of bytes multiplied by the quotient of the number of bits in a storage unit divided by eight. For example, if 100 bytes are required, specify:

```
for T'STORAGE_SIZE use 100 * (System.STORAGE_UNIT/8);
```

This provides a consistent approach to allocating storage as bytes. Confusion could otherwise result when one implementation uses bytes as the storage unit, and another uses words (two bytes).

Guide(07): If **pragma** "PACK" is used, verify that it is supported in the same way on the original and new target implementations.

Guide(08): Dependence on the "STORAGE_ERROR" exception is not advised. It is unclear what resources are available to the application after such an error has been detected. If its use is required, care must be taken to force deallocation of sufficient storage as the first portion of the handler. This may be achieved though explicit unchecked_deallocation requests, or by leaving the scope of a block to free up both heap and stack space.

5.3 Performance Issues

Guide(09): If **pragma** "SUPPRESS" is used, verify that it has an effect and that the performance improves.

Guide(10): Use of **package** "MACHINE_CODE" should be strictly controlled and delineated by configuration management. Good documentation and brevity of the subprograms is extremely important.

Guide(11): If **pragma** "INLINE" is used, insure that the compiler generates a warning if the desired effect is not achieved. The disassembled programs should be examined to verify the quality of the inlined subprograms as well as general code generation quality.

Guide(12): Be aware that some implementations have substantial overhead associated with the elaboration of block statements. Use them with some discretion.

Guide(13): Exception propagation overhead varies considerably among implementations. If possible do not expect fast exception propagation. If this is required, then the specific performance required must be documented in the Transporting Manual. Avoid the use of user defined exceptions as flags. More consistent timing is achieved by the direct use of variables for (boolean) flags, although the explicit checking of flags may consume more execution time in the typical case. Exceptions should be used for truly exceptional conditions (things that are never expected to occur, but may because of hardware failures or software design errors).

Guide(14): Do not use implementation-defined exceptions. [3] There is no consistency among different compilation systems.

Guide(15): Aggregate assignments both in elaboration and execution code vary widely between implementations. Establish benchmarks appropriate for your application that *indicate major changes in time/space* for these operations.

Guide(16): Are exceptions raised where response to the exception must be handled in real-time? Be aware that exception handling overhead can very greatly as a function of compiler and/or linker switches that support exception trace-back capability.

Guide(17): Measurements should be done on the execution time of every procedure as well as each rendezvous. Best, worst, typical, and average times should be recorded for each item. These should be preserved and compared during the transporting effort. All tests should be run with the compiler and linker options which will be used for both testing and system delivery. Note that turning the optimizer on has frequently been observed to slow down some portions of the code, and may even result in a net performance degradation. For example, this can occur if temporaries are generated to save index calculations which are not referenced more than once.

5.4 Tasking Issues

Guide(18): Selection of the scheduling algorithm is totally non- standard. Generally all useful implementations support as a default a "run 'till blocked (RTB)" scheduler that supports at least 16 levels of priority. This implies that a task will run until it is suspended waiting for I/O, a rendezvous, a **delay**, or until preempted by a higher priority task. Many implementations also support various flavors of time slicing, including a mixture of time-sliced and RTB. It is advisable to keep the tasking requirements on the runtime as simple as possible, since taking advantage of the more complex features may unnecessarily limit transportability if they are not absolutely required. Always explicitly state the priority of each task, including the main procedure. Use a configuration file to define all of the

priority constants, which should be specified in terms of priority'last and predecessors of each other.

Example:

```
with System; use System;

package Config is

-- Priorities of all tasks in order from highest priority to lowest...

graphics_priority : constant PRIORITY := PRIORITY'last;
operator_priority : constant PRIORITY := PRIORITY'PRED(graphics_priority);
rocket_priority   : constant PRIORITY := PRIORITY'PRED(operator_priority);

   ...
   ...
```

Guide(19): Do not depend on task activation to occur at the higher priority of the activator or the task being activated. This restriction may change in Ada9x (the future revision of the language).

Guide(20): It is advisable that each task have an "**entry** Synchronize" that is signaled as a consequence of the main program execution. Each task would "**accept** Synchronize" as their first statement. This allows much more user control over the "start up" of tasks declared in the outermost level of a library **package**. Since these tasks are activated in order of elaboration, they tend to start execution in an order defined by the compiler. In any case, liberal use of **pragma** "ELABORATE" is suggested to insure the library-level tasks do not call **package** procedures before their bodies are elaborated.

Guide(21): If the main program terminates via an exception handler, it should **abort** any library-level tasks. This insures that the library tasks will terminate (assuming **abort** is implemented asynchronously). Generally, a program with library-level tasks should only provide initialization code in the main procedure.

Guide(22): Be aware that task abortion may or may not take place immediately. If it is important that a particular implementation approach is depended upon, then this must be clearly stated in the design. Tasks may continue to execute indefinitely even after being **aborted** on certain implementations.

Guide(23): Only specify one task per **abort** statement. This forces an explicit order of abortion.

Guide(24): Dependence on rendezvous optimizations is an unfortunate reality for real-time programs. Obviously, limiting the rate at which rendezvous occur is a design goal to reduce the dependence on compiler optimizations. The use of implementation defined **pragmas** to indicate specific optimizations should be allowed when absolutely necessary. Often, similar but not identical **pragmas** are supported by several implementations. An example of this is the execution of an interrupt **accept** body directly without requiring a full task context switch. This can significantly reduce both the latency and overhead associated with

processing high-rate interrupts. On the other hand, if the additional processing of a conventional context switch can be tolerated, it is preferable to omit specifying any optimization. If optimization is required, insure that the restrictions indicated for the **pragma** are observed. There is generally no checking performed by the compiler to insure the code meets the restrictions, and erratic behavior generally results if they are violated.

Guide(25): Do not depend on "**delay** 0.0" to result in a scheduling event. That is, for a Run 'till Blocked scheduler, many implementations will treat "**delay** 0.0" as a request to put the current task on the end of the "ready" list, allowing other tasks to potentially execute. However there is no assurance that this is done for all implementations, and some may optimize very small **delay**s into tight timing loops or clock polling loops. Unless this dependence is clearly indicated, a "working" program may cause task starvation in a new runtime environment during heavy loading conditions.

Guide(26): Do not expect **delay** resolutions of less than 5ms. Most implementations allow configuration of the **delay** resolution, but at the expense of receiving and processing a timer interrupt at the minimum interval. This will usually result in excessive overhead if the required resolution is below a few milliseconds.

Guide(27): If more than one **delay** alternative is specified in a selective wait, do not depend on which one will be taken in cases where their values are (nearly) the same.

Guide(28): Use **pragma** "SHARED" for any scaler variables accessed by more than one task. If the **pragma** is not supported for that type, either change the program so the shared variable is not necessary, or manually make accesses to the variable atomically by disabling/enabling interrupts. Be sure that no exception is possible during the period interrupts are disabled. When accessing non-scaler types, make sure that reads and writes are performed as atomic actions by viewing the generated code.

5.5 Interrupt Processing Issues

Guide(29): Do not allow a task that contains interrupt entries to terminate prior to disabling the interrupt source. Failure to do so may allow the interrupt to arrive and subsequently execute the corresponding **accept** body after the task has terminated. This includes the use of a **terminate** alternative within the interrupt handler task.

Guide(30): For tasks with interrupt entries, always reserve the highest software priorities for these tasks. This will help to insure that the task can return to the **accept** statement without being preempted by other software tasks. Note that although the **accept** body is executed at the "interrupt priority", after the **accept** statement the priority resumes to the previous software priority. This is true even if the only statement outside the **accept** statement is a "**loop ... end loop**;". Many implementations optimize away this construct, yet this is not guaranteed. The software priority should be assigned according to the urgency in which the next interrupt could arrive in relation to other interrupt tasks.

Guide(31): Generally it is not recommended to perform an unconditional rendezvous within an interrupt **accept** body. If the **accept** body is suspended during an interrupt, the state of the hardware may preclude other interrupts from being serviced. Most implementations that support interrupt entries also provide some capability to signal other "support" tasks from within an interrupt task. Read the documentation carefully since they may also require certain restrictions. As always, use the simplest approach available that will meet your

requirements. Use of a parameterless conditional entry call is recommended. If shared data is modified by an interrupt task, the corresponding interrupt should be disabled while accesses are made from other tasks.

5.6 Numeric Issues

Guide(32): Do not depend on "NUMERIC_ERROR". Wherever "NUMERIC_ERROR" is expected, always use "CONSTRAINT_ERROR | NUMERIC_ERROR". The language maintenance process has established an approved language interpretation (#387) that states:

```
"Wherever the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement),
 CONSTRAINT_ERROR should be raised instead.

This interpretation is non-binding."
```

The fact that the interpretation is non-binding, means that it is implementation dependent.

Guide(33): Always make sure that each calculation can be performed within the range of the base type of the operands. Some implementations may perform intermediate calculations with a larger range than required by the Ada language standard, but small variations in the runtime environment could change this (including an optimize phase). For example:

```
type SMALL_TYPE is range 1..100;
VAR1 : SMALL_TYPE := 4;
VAR2 : SMALL_TYPE := 80;

VAR1 := VAR1 * VAR2 / 10;
```

The above assignment may work for a while producing the correct result of 32, but after an optimize phase or transporting, the intermediate result of 320 may cause an exception. Define the base type to be large enough to support all calculations, then use subtypes for object declarations. See RM 11.6(6)

Guide(34): Provide sufficient explicit checks to insure that overflow conditions do not occur. Do not depend on the Ada implementation to detect an overflow condition. Besides the fact that it may be necessary to suppress checks to obtain performance, some implementations simply do not support overflow detection.

Guide(35): Do not use equality comparisons for floating point types. Instead, subtract the two operands and compare the difference to be less than some allowable value. This difference should exceed the greater model interval of the types. See RM 4.5.7(10).

Guide(36): If errors are possible due to the rounding algorithm used when converting real types to integer types, code must be included to provide explicit rounding. This may take the form of a **generic** "Round" **function** that is used whenever converting real types to an integer type. If possible, avoid dependence on a particular rounding convention. See RM 4.6(7).

Guide(37): If errors are possible due to the rounding algorithm used for the real predefined operators types, the type should be defined with additional digits of accuracy, or additional code should be added to compensate for accumulated error. Note that the attribute "MACHINE_ROUNDS" is more helpful when it is "TRUE", because real operators may still perform rounding in some cases when it is "FALSE".

Guide(38): Do not use the predefined numeric types. Define a **package** "Types" for the application dependent standard types based strictly on the ranges and accuracies required (for example: "type ROAD_LENGTH_TYPE is range 0..37000;"). Careful attention to correctly specifying the required accuracy and range of numeric types is essential for transportability. Implementations may compute different results, but are required to be within the accuracy constraints imposed on the real types. This implies that if the real types are defined properly, the application should operate correctly on all configurations which can support those types. Implementations that cannot support the type are obliged to reject the program during compilation. Note that the predefined type "STRING" and many of the I/O subprograms use the predefined "INTEGER" type. You will have to convert from the application integer types to "INTEGER" if you use them.

Guide(39): Most implementations do not support fixed point type length clauses for T'Small that are not a power of two. Inform the hardware designers of this limitation and request that hardware values be supplied as a power of two. For example, temperatures should not be in tenths of a degree (0.1) but rather in eighths (0.125) or sixteenths (0.0625). One alternative is to read the values as integers, and then immediately convert them to an appropriate fixed point type while applying the necessary scale factor. This will of course reduce the accuracy of the values somewhat. The other alternative is to implement a scaled integer arithmetic **package**, which is not easily done and can result in performance problems.

Guide(40): It must not be assumed that a static expression is evaluated with the same accuracy than that of the model numbers of a particular real type. An implementation is not required to produce the same value for a static expression by compile-time evaluation on a host computer system as it would produce for the same expression at runtime on the target computer system. These values are only required to be in the same model number interval. The consequence of this potential imprecision makes the results of comparisons implementation dependent.

5.7 Subprogram Issues

Guide(41): Since parameters of composite types may be passed by reference, multiple access paths are a potential for these objects. Insure that every composite type passed as a parameter is never passed more than once during the invocation of a subprogram (or **accept** body) and is not accessed as a global variable within the subprogram or **accept** body. This is especially true of recursive subprograms, since the behavior depends on whether or not the object is copied or simply a reference is passed. Special care must be taken when writing generic software for which the type is not known.

Guide(42): If return types from functions are unconstrained, verify that any storage created for such objects is always deallocated after the function call.

Guide(43): Always assign a value to **out** mode parameters for procedures. See RM 6.2(5)

Guide(44): Most embedded systems do not execute commands which invoke Ada programs, however it is possible that some control over program invocation may exist. For example, an embedded system may have a general purpose network driver which can accept commands, load an Ada program and execute it. If command line (invocation) parameters are used for the main program, application access to them should be strictly hidden by an application defined subprogram.

Guide(45): Functions should not have side-effects. This eliminates the vast majority of situations where order dependencies occur. If they are necessary, they should be documented to clearly indicate all possible side-effects. Such a function should not be used in a statement with any of the objects it modifies, or with any other function (including itself) which modifies the same objects. If they must be in the same statement, insure that if any valid order is chosen, the results will be correct with respect to the accuracy requirements and that the execution timing is essentially the same. Note that even though all possible results may appear to be "sufficiently correct", it is still better to eliminate the order dependency. The slight difference in results may effect the flow of the program, and therefore result in latent design errors becoming active.

5.8 Input/Output Issues

Guide(46): Assume as little as possible about the I/O support available. [3] If file storage is required in the application, Ada source code for the file system should be available and should be based on common (i.e. 512-byte) block oriented access schemes that are usually supported directly by the hardware. Most bare targets have little if any I/O support, especially for file oriented I/O.

Guide(47): If temporary files are created, use an application program to generate a unique name rather than using a null string as the file name. Always explicitly delete these files prior to termination. If this is impractical and standard temporary files must be used (i.e. null string names), insure that the system automatically deletes them. If not, provide some method to achieve the same effect. The status of a temporary file is unknown after it is closed. That is, some implementations may delete it immediately. Use the "Reset" **procedure** to access information in temporary files. The "Name" **function** should not be used to obtain the name of a temporary file since it may raise "USE_ERROR".

Guide(48): Requiring interchange of any files between two runtime environments has a high probability of making a program non- transportable. This is because the I/O packages instantiated for types may have different representations as they are stored in the file. The problem generally occurs when a database is maintained by a program. Note that the runtime environment change could be caused by any recompilation of the program. Since the compiler is free to choose the default representations of data objects, a new optimize phase may result in a different representation for a type. This would cause the program to misinterpret the data written out by a previous execution of the same program. What is even more likely is the simultaneous sharing of a database by different releases of the same program or even different hardware environments. This is typical of a factory automation situation where data logging and manufacturing parameters are maintained over a period of time by a central data base and updated and referenced by several work stations connected by a network. If file interchange is required between the original program and the transported program, it is suggested that a standard file format be specified and representation clauses used to help achieve the standard format. Another approach is to write all data in a standard ASCII text format, although the overhead associated with this

approach is usually too great for a real-time program. If this approach is used, avoid use of the "END_OF_LINE", "PAGE", and "END_OF_FILE" delimiters. Instead, substitute with application defined representations for these terminators. Finally, it may also be possible to provide conversion programs that provide one-time or on-demand translation of a shared data base to the required format.

Guide(49): Be especially cautious of performing I/O on access types or unconstrained types. Access types will typically not be of value from one program execution to the next, since they are frequently memory addresses which are no longer meaningful after the program has terminated. Unconstrained types must provide constraint information in addition to the data when transferring between files and memory. The format of this constraint information is not standard among implementations and there is no way for an application program to specify a particular format. See RM 14.1(1) and 14.1(7).

Guide(50): Concurrent sharing of external files is extremely implementation dependent. If possible, design the program so this is not necessary. If sharing is required, document the requirement and provide details as to how the file(s) are shared. This includes a single task opening the same external file twice, more than one task opening the same external file, or more than one program opening the same external file. The same holds true for deleting shared files. See RM 14.1(13). It is recommended that a "monitor" **task** (or program) be used to provide access to shared files.

Guide(51): If possible, use file names that begin with an alpha character and contain only alphanumeric characters. Ideally, file names should be kept to eight characters unless confusion can occur as to the purpose of the file. Avoid use of the "FORM" parameter except when absolutely necessary.

Guide(52): Do not depend on the raising of "DATA_ERROR" while reading files. Implementations are allowed to omit checks for data correctness. See RM 14.2.2(4) and 14.2.4(4).

Guide(53): Always close files prior to program termination. See RM 14.1(7).

Guide(54): Do not depend on a specific representation for "LINE", "PAGE", and "FILE" terminators.

Guide(55): For interactive devices and "GET_LINE", be aware that some implementations wait for additional characters to be entered after the line terminator to check for a page or file terminator. Verify the operation of the target runtime system to ascertain its characteristics. This additional wait is considered an undesirable approach to text input and the vendor should be advised of the problem. Note that this is separate from waiting when an explicit call to "End_Of_File" or "End_Of_Page" is made. Implementations have little choice but to wait until some input is available for these functions.

Guide(56): Many systems buffer input and output. This may effect the timing and the presentation order of data (especially when two tasks share a console). Devices shared by tasks should have their access serialized by providing a task as an interface to the device. Any buffering mechanism that is required for proper program execution must be documented.

5.9 Other Issues

Guide(57): Avoid use of implementation defined attributes, types, and exceptions. For example, if a 16-bit "WORD" type is defined in **package** "System", define an identical type in the application's "Types" **package** to use in its place. Using implementation dependent aspects of **package** "System" is a very common mistake which causes serious transportability problems. It can even cause difficulty with the use of Ada-PDL processors, since they may not allow redefinition of **package** "System".

Guide(58): Use only ISO seven-bit coded characters. Some implementations allow the 8-bit character set within comment fields, but this is not accepted by all implementations. Alternatively, provide a preprocessor that removes these characters or substitutes some corresponding string prior to compilation.

Guide(59): Restrict representation clauses for enumeration literals to unsigned integers.

Guide(60): Always initialize a variable prior to referencing it. This is simply good programming practice. Be careful to realize that passing a variable as an **in** or **in out** parameter is equivalent to referencing, even if the variable will not be referenced prior to assignment in the subprogram. This practice does not imply that all variables must be provided with an initial value (at the time of declaration) however.

Guide(61): Document the bit ordering used for all record representation clauses. These are not standard and are likely to have an impact during transporting a program. It may be desirable to develop a source code translator that will translate from one bit ordering to another, provided component clauses are defined in a uniform way. Most microprocessor hardware conventions refer to bit zero (0) as the least significant bit. Therefore it is helpful to document the application requirements in terms of least significant and most significant bits rather than using terms such as "left" or "right" which are meaningless in this context.

Guide(62): Do not reference generated names for implementation dependent record components. See RM 13.4(8).

Guide(63): Whenever practical, isolate implementation dependencies within separate compilation units. This modularization helps to identify dependencies and facilitates modifications to them. Use the **package** facility to encapsulate these implementation dependencies. [11] This is the preferred technique for isolating implementation dependencies when you must use them.

Guide(64): Provide fully expanded names for all objects not defined in the immediate compilation unit. This does not include references to functions defined by operators or objects declared by the Ada language standard. Be careful to expand names of objects declared in **package** "System", but not explicitly identified in the Ada language standard. Also, references to objects that are declared in ancestor units (parents of subunits) which are immediately visible should still be fully expanded. This will not directly effect transportability but is often essential to program comprehension and maintenance. Therefore it also is beneficial to transporting programs that require some modification.

Guide(65): Avoid use of languages other than Ada. Mixing language creates at least two problems: the transportability of the other language(s), and transportability of the conventions and interfaces between Ada and the other language(s).

Guide(66): "Unchecked_Conversion" should only be used for statically constrained types of the same size. If sizes are different, create a record type with the same size of the larger type, and provide explicit values for the additional fields when converting between the large type and the created record type.

For example:

```
type LONG_CONVERT_TYPE is
  record
    LOW_WORD  : Types.WORD;     -- 16 bits each
    HIGH_WORD : Types.WORD;
  end record;

for LONG_CONVERT_TYPE use
  record
    LOW_WORD  at 0 range 0..15;
    HIGH_WORD at 0 range 16..31;
  end record;
  for LONG_CONVERT_TYPE'size use 32;

function Word_To_Long is new
  Unchecked_Conversion(LONG_CONVERT_TYPE,Types.LONG_WORD)   -- same size

  WORD_DATA    : Types.WORD;                  -- 16 bits
  LONG_DATA    : Types.LONG_WORD;             -- 32 bits
...
  LONG_DATA := Word_To_Long((WORD_DATA,0));   -- always zero fill high word!
                                              -- (no sign extension)
```

Guide(67): Do not reference "System.MEMORY_SIZE". This has no consistent definition among implementations. What might be more useful is to define functions which return the amount of storage currently available for allocators (heap) or for subprogram invocation (stack). This might be helpful in allowing programs to take advantage of available memory for allocating more objects or deeper nesting of recursive subprograms. These functions would most likely require modification of the runtime to implement however. This leaves embedded systems no transportable way to respond to the amount of memory available.

Guide(68): Modifications to the vendor supplied runtime must be clearly documented and categorized as application related or target processor related. It is generally preferred to make any modifications in the form of subprograms that are called by the vendor runtime rather than making changes directly to the runtime code. This of course is only practical if the runtime is sufficiently modular.

Guide(69): Add a field in the documentation template for each **package**, **procedure**, **function**, and **task** indicating if any non-transportable features are present. In addition, for each non-transportable feature employed, provide a detailed description of the expectations for that feature. If possible, this comment should be extractable by a tool for placement in the Transporting Manual.

6. Summary

The ability to transport programs among different processor technologies is essential in the maintenance of embedded systems. The benefit of being able to upgrade processor hardware and/or compiler technology over the lifetime of embedded systems is substantial. Not only can processing throughput be improved, which is often necessary to add software capability, but it can reduce the difficulty in obtaining parts for obsolete technology. By extending the useful life of these systems, cost savings can be achieved while modernization makes the system meet new requirements (threats) much more quickly. Furthermore, transportability of embedded software is often required for the reuse of software components which can be shared among similar embedded application areas.

This handbook provides guidelines to improve the transportability of real-time embedded applications software. Although it is not practical to achieve 100% transportability, it is reasonable to obtain sufficient processor independence so that transporting to higher performance targets is cost effective. This ability is essential if target processor selection is deferred until system integration time.

Although implementation dependencies are generally to be avoided, when specific characteristics are required of the compilation system it is preferred to explicitly state them in the source code. They typically take the form of representation clauses which force a particular representation rather than depending on the compilation system's default. In this way, other compilation systems can either comply with the request or reject the program. This reduces the likelihood that the transported program will execute incorrectly in very subtle ways.

7. References

[1] Reference Manual for the Ada Programming Language.ANSI/MIL-STD-1815A-1983, American National Standards Institute, Inc., 1983.

[2] Nissan, Wallis, Wichmann and others, "Ada-Europe Guidelines for the Portability of Ada Programs", Ada Letters,ACM SIGAda, Volume I, Number 3, March, April 1982.

[3] F. Pappas, Ada Portability Guidelines, SofTech Inc., Waltham, MA, March 1985, DTIC/NTIS #AD-A160 390.

[4] J. Goodenough, and others, Ada Reusability Guidelines, SofTech Inc., Waltham, MA, April 1985, DTIC/NTIS #AD-A161 456.

[5] E. R. Matthews, "Observations on the Portability of Ada I/O", ACM Ada Letters VII(5), pp.100-103 (September, October 1987).

[6] Peter Freeman, Tutorial: Software Reusability, Computer Society Press, 1987.

[7] "Ada Compiler Validation Procedures and Guidelines", Ada Joint Program Office, Ada Letters, ACM SIGAda, Volume VII, Number 2, March, April 1987.

[8] "A Framework for Describing Ada Runtime Environments", Ada Runtime Environment Working Group of SIGAda, October 15, 1987.

[9] "Catalogue of Ada Runtime Implementation Dependencies", CECOM, Center for Software Engineering, CIN: C02092JB0001, 15 February 1989, and ACM SIGAda Ada Runtime Environment Working Group, December 1, 1987, version 2.0.

[10] J. Ichbiah, et al, Rationale for the Design of the Ada Programming Language, ACM SIGPLAN Notices, 14(6), Part B (June 1979).

[11] "Ada Style Guide", Software Productivity Consortium, Reston, VA, SPC-TR-88-003, Version 1.0, February 1988.

8. Glossary

ARTEWG: The Ada RunTime Environment Working Group, is a group sponsored by the Association for Computing Machinery (ACM), Special Interest Group for Ada (SIGAda), whose purpose is to address the problems encountered in runtime environments.

PIWG: The Performance Issues Working Group, is a group sponsored by the Association for Computing Machinery (ACM), Special Interest Group for Ada (SIGAda), whose purpose is to write benchmark programs which can be executed on different Ada compilation systems and provide performance information.

Target Architecture: The computer architecture used for execution of object code generated by an Ada compiler. [7]

9. Appendix A - Implementation Tests

The tests in this appendix will need to be customized to compile on each prospective target. This is because they are constructed almost entirely of implementation dependencies for the purposes of determining which ones are supported by an implementation. In many cases, the only output from a test will be compiler diagnostics indicating that a feature is not supported. In these cases, the user can usually comment the offending statement, or make small modifications to allow the program to compile.

These tests may be used to help determine the implementation approach used by both the initial target and the new retargeted environment. This information is useful to isolate problems that are not otherwise obvious. The results of the tests should be documented in the Transporting Manual. If problems occur during the transporting effort, the differences in implementation approaches identified by the tests can provide a good starting point for determining the cause of the problem.

```
--************************************************************************
-- TST_LEN_PACK - This procedure tests the implementation of
--   the SIZE representation clause and the pragma PACK for simple
--   types and arrays.
--   The tests use the following simple types:
--    A BOOLEAN set to 8 bits, a BOOLEAN set to 1 bit, an unsized
--     4-bit INTEGER, a 4-bit INTEGER set to 4 bits, an unsized
--     3-bit INTEGER, and a 3-bit INTEGER set to 3 bits.
--    Three arrays are declared for each simple type, one that is
--     is not packed, one that is packed, and one that is unpacked
--     but sized to the minimal possible size.
--    The size (from the SIZE attribute) is printed for all 24 array
--       types.
--    Two questions concerning fixed point types are also given as
--     declarations.  Can the size of a fixed point type be set to
--     half the size of the size of the implementation's INTEGER type?
--     Can 'SMALL for a fixed point type be set to a non-power of two?
--
--   Author: R.W. Sebesta
--   Date:   August, 1988
--
with TEXT_IO;
use TEXT_IO;
procedure TST_LEN_PACK is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type SMALL_BOOL_TYPE is new BOOLEAN;
  for SMALL_BOOL_TYPE'SIZE use 8;
  type TINY_BOOL_TYPE is new BOOLEAN;
  for TINY_BOOL_TYPE'SIZE use 1;
  type SMALL_INT_1_TYPE is range 0..15;
  for SMALL_INT_1_TYPE'SIZE use 4;
  type SMALL_INT_2_TYPE is range 0..15;
  type TINY_INT_1_TYPE is range 0..7;
```

```
   for TINY_INT_1_TYPE'SIZE use 3;
   type TINY_INT_2_TYPE is range 0..7;

   type ARRAY_BOOL_1_TYPE is array (1..100) of SMALL_BOOL_TYPE;
   type ARRAY_BOOL_2_TYPE is array (1..100) of TINY_BOOL_TYPE;
   type ARRAY_SMALL_INT_1_TYPE is array (1..100) of SMALL_INT_1_TYPE;
   type ARRAY_SMALL_INT_2_TYPE is array (1..100) of SMALL_INT_2_TYPE;
   type ARRAY_TINY_INT_1_TYPE is array (1..100) of TINY_INT_1_TYPE;
   type ARRAY_TINY_INT_2_TYPE is array (1..100) of TINY_INT_2_TYPE;

   type ARRAY_BOOL_1_TYPE_P is array (1..100) of SMALL_BOOL_TYPE;
   type ARRAY_BOOL_2_TYPE_P is array (1..100) of TINY_BOOL_TYPE;
   type ARRAY_SMALL_INT_1_TYPE_P is array (1..100) of SMALL_INT_1_TYPE;
   type ARRAY_SMALL_INT_2_TYPE_P is array (1..100) of SMALL_INT_2_TYPE;
   type ARRAY_TINY_INT_1_TYPE_P is array (1..100) of TINY_INT_1_TYPE;
   type ARRAY_TINY_INT_2_TYPE_P is array (1..100) of TINY_INT_2_TYPE;

   pragma PACK (ARRAY_BOOL_1_TYPE_P);
   pragma PACK (ARRAY_BOOL_2_TYPE_P);
   pragma PACK (ARRAY_SMALL_INT_1_TYPE_P);
   pragma PACK (ARRAY_SMALL_INT_2_TYPE_P);
   pragma PACK (ARRAY_TINY_INT_1_TYPE_P);
   pragma PACK (ARRAY_TINY_INT_2_TYPE_P);

   type ARRAY_BOOL_1_TYPE_S is array (1..100) of SMALL_BOOL_TYPE;
   type ARRAY_BOOL_2_TYPE_S is array (1..100) of TINY_BOOL_TYPE;
   type ARRAY_SMALL_INT_1_TYPE_S is array (1..100) of SMALL_INT_1_TYPE;
   type ARRAY_SMALL_INT_2_TYPE_S is array (1..100) of SMALL_INT_2_TYPE;
   type ARRAY_TINY_INT_1_TYPE_S is array (1..100) of TINY_INT_1_TYPE;
   type ARRAY_TINY_INT_2_TYPE_S is array (1..100) of TINY_INT_2_TYPE;
   for ARRAY_BOOL_1_TYPE_S'SIZE use 100;
   for ARRAY_BOOL_2_TYPE_S'SIZE use 100;
--   for ARRAY_SMALL_INT_1_TYPE_S'SIZE use 400;
--   for ARRAY_SMALL_INT_2_TYPE_S'SIZE use 400;
--   for ARRAY_TINY_INT_1_TYPE_S'SIZE use 300;
--   for ARRAY_TINY_INT_2_TYPE_S'SIZE use 300;

   type FIXED_TYPE is delta 0.125 range -5.0..5.0;
--   for FIXED_TYPE'SIZE use INTEGER'SIZE / 2;
--   for FIXED_TYPE'SMALL use 0.01;

   begin
   PUT ("Size of an unpacked array of 100 8-bit BOOLEAN elements is:");
   PUT (ARRAY_BOOL_1_TYPE'SIZE);   NEW_LINE;
   PUT ("Size of an unpacked array of 100 single-bit BOOLEAN elements is:");
   PUT (ARRAY_BOOL_2_TYPE'SIZE);   NEW_LINE;
   PUT ("Size of an unpacked array of 100 4-bit sized INTEGER elements is:");
   PUT (ARRAY_SMALL_INT_1_TYPE'SIZE);   NEW_LINE;
   PUT ("Size of an unpacked array of 100 4-bit unsized INTEGER elements is:");
   PUT (ARRAY_SMALL_INT_2_TYPE'SIZE);   NEW_LINE;
   PUT ("Size of an unpacked array of 100 3 bit sized INTEGER elements is:");
```

```
PUT (ARRAY_TINY_INT_1_TYPE'SIZE);   NEW_LINE;
PUT ("Size of an unpacked array of 100 3-bit unsized INTEGER elements is:");
PUT (ARRAY_TINY_INT_2_TYPE'SIZE);   NEW_LINE;
NEW_LINE;

PUT ("Size of a packed array of 100 8-bit BOOLEAN elements is:");
PUT (ARRAY_BOOL_1_TYPE_P'SIZE);   NEW_LINE;
PUT ("Size of a packed array of 100 single-bit BOOLEAN elements is:");
PUT (ARRAY_BOOL_2_TYPE_P'SIZE);   NEW_LINE;
PUT ("Size of a packed array of 100 4-bit sized INTEGER elements is:");
PUT (ARRAY_SMALL_INT_1_TYPE_P'SIZE);   NEW_LINE;
PUT ("Size of a packed array of 100 4-bit unsized INTEGER elements is:");
PUT (ARRAY_SMALL_INT_2_TYPE_P'SIZE);   NEW_LINE;
PUT ("Size of a packed array of 100 3-bit sized INTEGER elements is:");
PUT (ARRAY_TINY_INT_1_TYPE_P'SIZE);   NEW_LINE;
PUT ("Size of a packed array of 100 3-bit unsized INTEGER elements is:");
PUT (ARRAY_TINY_INT_2_TYPE_P'SIZE);   NEW_LINE;
NEW_LINE;

PUT ("Size of a sized unpacked array of 100 8-bit BOOLEAN elements is:");
PUT (ARRAY_BOOL_1_TYPE_S'SIZE);   NEW_LINE;
PUT ("Size of a sized unpacked array of 100 single-bit BOOLEAN elements is:");
PUT (ARRAY_BOOL_2_TYPE_S'SIZE);   NEW_LINE;
--  PUT ("Size of a sized unpacked array of 100 4-bit sized INTEGER elements is:");
--  PUT (ARRAY_SMALL_INT_1_TYPE_S'SIZE);   NEW_LINE;
--  PUT ("Size of a sized unpacked array of 100 4-bit unsized INTEGER elements is:");
--  PUT (ARRAY_SMALL_INT_2_TYPE_S'SIZE);   NEW_LINE;
--  PUT ("Size of a sized unpacked array of 100 3-bit sized INTEGER elements is:");
--  PUT (ARRAY_TINY_INT_1_TYPE_S'SIZE);   NEW_LINE;
--  PUT ("Size of a sized unpacked array of 100 3-bit unsized INTEGER elements is:");
--  PUT (ARRAY_TINY_INT_2_TYPE_S'SIZE);   NEW_LINE;
end TST_LEN_PACK;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_COLL - This procedure tests the STORAGE_SIZE
--   representation clause for collections for access types.
--   Three access types are defined, one with a STORAGE_SIZE
--   specified.  The type that is tested last is to used to
--   determine whether there is any interference among collections.
--
--   Author: R.W. Sebesta
--   Date:   July, 1988
--
with TEXT_IO;
use TEXT_IO;
procedure TST_COLL is
  type BIG_INT is range 0..1000000;
  package BIG_INT_IO is new INTEGER_IO(BIG_INT);
  use BIG_INT_IO;
  type INT_16 is range -32768..32767;
  for INT_16'SIZE use 16;
  type INT_1_PTR_TYPE is access INT_16;
  type INT_2_PTR_TYPE is access INT_16;
  for INT_2_PTR_TYPE'STORAGE_SIZE use 10000;
  type INT_3_PTR_TYPE is access INT_16;

  PTR_1         : INT_1_PTR_TYPE;
  PTR_2         : INT_2_PTR_TYPE;
  PTR_3         : INT_3_PTR_TYPE;
  NUM_OBJECTS   : BIG_INT;

  begin
   DEF_SIZE_1:
    begin
    PUT ("The default size for an INT_16 collection is:");
    PUT (INT_1_PTR_TYPE'STORAGE_SIZE);   NEW_LINE;
    NUM_OBJECTS := 0;
    for COUNT in 1..1000000 loop
      PTR_1 := new INT_16;
      NUM_OBJECTS := NUM_OBJECTS + 1;
      end loop;
    exception
      when STORAGE_ERROR =>
        PUT ("Maximum number of INT_16 objects with default STORAGE_SIZE is:");
        PUT (NUM_OBJECTS);   NEW_LINE;
    end DEF_SIZE_1;

  SET_SIZE:
    begin
    PUT ("Size of a collection for INT_16 objects that is set to 10000 is:");
    PUT (INT_2_PTR_TYPE'STORAGE_SIZE);   NEW_LINE;
    NUM_OBJECTS := 0;
    for COUNT in 1..1000000 loop
```

```
    PTR_2 := new INT_16;
    NUM_OBJECTS := NUM_OBJECTS + 1;
    end loop;
  exception
    when STORAGE_ERROR =>
      PUT ("Maximum number of INT_16 objects with STORAGE_SIZE = 10000 is:");
      PUT (NUM_OBJECTS);   NEW_LINE;
  end SET_SIZE;

DEF_SIZE_2:
  begin
  PUT ("Default size of a collection for INT_16 after all storage is used is:");
  PUT (INT_3_PTR_TYPE'STORAGE_SIZE);  NEW_LINE;
  NUM_OBJECTS := 0;
  for COUNT in 1..1000000 loop
    PTR_1 := new INT_16;
    NUM_OBJECTS := NUM_OBJECTS + 1;
    end loop;
  exception
    when STORAGE_ERROR =>
      PUT ("Maximum number of INT_16 objects with default STORAGE_SIZE is:");
      PUT (NUM_OBJECTS);   NEW_LINE;
  end DEF_SIZE_2;
end TST_COLL;
```

```
  NUM_INTS: INTEGER := 0;
  begin
  accept TST_HEAP_ENT do
    for COUNT in 1..1000000 loop
      INT_PTR := new INT_16;
      NUM_INTS := NUM_INTS + 1;
      end loop;
    end TST_HEAP_ENT;
    exception
      when STORAGE_ERROR =>
        PUT ("Number of INT_16 objects allocated before STORAGE_ERROR:");
        PUT (NUM_INTS);  NEW_LINE;
  end TST_HEAP_TYPE_1;

task body TST_STK_TYPE_1 is
  NUM_PROCS   : INTEGER := 0;
  procedure TAKE_SPACE is
    type LIST_TYPE is array (1..100) of INT_16;
    LIST      : LIST_TYPE;
    begin
    NUM_PROCS := NUM_PROCS + 1;
    TAKE_SPACE;
    exception
      when STORAGE_ERROR =>
        PUT ("Procedure TAKE_SPACE ran out of space after");
        PUT (NUM_PROCS);  PUT (" calls (100 INT_16s each)");  NEW_LINE;
    end TAKE_SPACE;

  begin
  accept TST_STK_ENT do
    TAKE_SPACE;
    end TST_STK_ENT;
  end TST_STK_TYPE_1;

begin

PUT_LINE ("Test run of the TST_STK_2 task (STORAGE_SIZE = 20000)");
PUT ("Initial task STORAGE_SIZE is:");
PUT (TST_STK_2'STORAGE_SIZE);   NEW_LINE;
TST_STK_2.TST_STK_ENT;

PUT_LINE ("Test run of the TST_STK_1 task (without rep clause)");
PUT ("Initial task STORAGE_SIZE is:");
PUT (TST_STK_1'STORAGE_SIZE);   NEW_LINE;
TST_STK_1.TST_STK_ENT;

PUT_LINE ("Test run of the TST_HEAP_2 task (STORAGE_SIZE = 20000)");
PUT ("Initial task STORAGE_SIZE is:");
PUT (TST_HEAP_2'STORAGE_SIZE);   NEW_LINE;
TST_HEAP_2.TST_HEAP_ENT;
```

```
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_TSK_STOR - This procedure tests the use of the STORAGE_SIZE
--  representation clause and the STORAGE_SIZE attribute for tasks.
--  Four tests are attempted.  If any of them uses all available
--  storage, the subsequent tests are obviously not made.  In these
--  cases, the order of the tests can easily be changed--the four
--  are instigated by code sequences at the end of the procedure
--  TST_TSK_STOR.
--
--  The four tests are:
--  1. Set STORAGE_SIZE for a task to 20000 and cause it to allocate
--     stack space until the STORAGE_ERROR is raised.
--  2. Same as 1, except without STORAGE_SIZE set.
--  3. Set STORAGE_SIZE for a task to 20000 and cause it to allocate
--     heap space until the STORAGE_ERROR is raised.
--  4. Same as 3, except without STORAGE_SIZE set.
--
--  Author: R.W. Sebesta
--  Date:   August, 1988
--
with TEXT_IO;
use TEXT_IO;
procedure TST_TSK_STOR is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;

  task type TST_HEAP_TYPE_1 is
    entry TST_HEAP_ENT;
    end TST_HEAP_TYPE_1;
  task type TST_STK_TYPE_1 is
    entry TST_STK_ENT;
    end TST_STK_TYPE_1;

  type TST_HEAP_TYPE_2 is new TST_HEAP_TYPE_1;
  for TST_HEAP_TYPE_2'STORAGE_SIZE use 20000;

  type TST_STK_TYPE_2 is new TST_STK_TYPE_1;
  for TST_STK_TYPE_2'STORAGE_SIZE use 20000;

  type INT_16 is range -32768..32767;
  for INT_16'SIZE use 16;

  TST_HEAP_1    : TST_HEAP_TYPE_1;
  TST_HEAP_2    : TST_HEAP_TYPE_2;
  TST_STK_1     : TST_STK_TYPE_1;
  TST_STK_2     : TST_STK_TYPE_2;

  task body TST_HEAP_TYPE_1 is
    type INT_PTR_TYPE is access INT_16;
    INT_PTR     : INT_PTR_TYPE;
```

```
PUT_LINE ("Test run of the TST_HEAP_1 task (without rep clause)");
PUT ("Initial task STORAGE_SIZE is:");
PUT (TST_HEAP_1'STORAGE_SIZE);   NEW_LINE;
TST_HEAP_1.TST_HEAP_ENT;
end TST_TSK_STOR;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_ENUM_TIME - This procedure determines the relative cost of
--  using representation clauses to force nonconsecutive values to
--  be used for enumeration types.  CALENDAR.CLOCK is used to time
--  a loop containing numerous references to arrays using an
--  enumeration types as their index types.  Also included in the
--  loop are uses of the attributes PRED and SUCC.  The loop, which
--  has 100000 repetitions, is repeated 10 times and the average
--  time is output.  The average of 10 repetitions is used to
--  avoid some of the inaccuracies of using CLOCK on a variety of
--  systems.
--
--  Author: R.W. Sebesta
--  Date:   August, 1988
--
with CALENDAR;
use CALENDAR;
with TEXT_IO;
use TEXT_IO;
procedure tst_enum_time is
  package FLT_IO is new FLOAT_IO (FLOAT);
  use FLT_IO;
  type ENUM_TYPE is (SUN, MON, TUE, WED, THU, FRI, SAT);
  for ENUM_TYPE use (SUN => -300, MON => -200, TUE => -100,
                     WED => 0, THU => 100, FRI => 200, SAT => 300);
  type LIST_TYPE is array (ENUM_TYPE) of INTEGER;
  LIST_1        : LIST_TYPE;
  LIST_2        : LIST_TYPE;
  TIME_1        : TIME;
  TIME_2        : TIME;
  TIME_USED     : DURATION;
  TOTAL_TIME    : FLOAT;
  TIME_USED_FLT : FLOAT;
  AVG_TIME      : FLOAT;
  begin
  TIME_1 := CLOCK;
  LIST_1(SUN) := 2;
  for INDEX in MON..SAT loop
    LIST_1(INDEX) := LIST_1(ENUM_TYPE'PRED(INDEX)) + 1;
    end loop;

  TOTAL_TIME := 0.0;

  for BIG_COUNT in 1..10 loop
    TIME_1 := CLOCK;
    for COUNT in 1..100000 loop
      for INDEX in SUN..SAT loop
        LIST_2(INDEX) := LIST_1(INDEX);
        end loop;
      for INDEX in SUN..SAT loop
```

```
      LIST_1(INDEX) := LIST_2(INDEX);
      end loop;
    for INDEX in MON..FRI loop
      LIST_1(ENUM_TYPE'SUCC(INDEX)) := LIST_2(ENUM_TYPE'PRED(INDEX)) - 1;
      LIST_1(ENUM_TYPE'PRED(INDEX)) := LIST_2(ENUM_TYPE'SUCC(INDEX)) + 1;
      end loop;
    end loop;
  TIME_2 := CLOCK;
  TIME_USED := TIME_2 - TIME_1;
  TIME_USED_FLT := FLOAT(TIME_USED);
  TOTAL_TIME := TIME_USED_FLT + TOTAL_TIME;
  PUT ("Time used is:");
  PUT (TIME_USED_FLT);  NEW_LINE;
  end loop;

PUT ("Total time for 10 iterations:");
PUT (TOTAL_TIME);  NEW_LINE;
AVG_TIME := TOTAL_TIME / 10.0;
PUT ("Average time per iteration:");
PUT (AVG_TIME);  NEW_LINE;

end TST_ENUM_TIME;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_STATIC_REC_ALIGN - This procedure tests an Ada implementation's
--  ability to provide for record alignment for static records.  The
--  static records are defined in the separate package STATIC_REC_DATA.
--  The four imported record objects, REC_1, REC_2, REC_3, and REC_4
--  are specified to have alignments of 2, 3, 4, and 8.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
with SYSTEM;
with UNCHECKED_CONVERSION;
with TEXT_IO;
use TEXT_IO;
with STATIC_REC_DATA;
use STATIC_REC_DATA;
procedure TST_STATIC_REC_ALIGN is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;,
  function CVT_ADDRESS_TO_INTEGER is new UNCHECKED_CONVERSION
       (SOURCE => SYSTEM.ADDRESS, TARGET => INTEGER);
  begin
--
-- Test for mod 2 alignment
--
  if CVT_ADDRESS_TO_INTEGER (REC_1'ADDRESS) mod 2 = 0
    then PUT_LINE ("Mod 2 alignment is OK");
    else PUT_LINE ("Mod 2 alignment is incorrect");
    end if;
--
-- Test for mod 3 alignment
--
--  if CVT_ADDRESS_TO_INTEGER (REC_2'ADDRESS) mod 3 = 0
--    then PUT_LINE ("Mod 3 alignment is OK");
--    else PUT_LINE ("Mod 3 alignment is incorrect");
--    end if;
--
-- Test for mod 4 alignment
--
  if CVT_ADDRESS_TO_INTEGER (REC_3'ADDRESS) mod 4 = 0
    then PUT_LINE ("Mod 4 alignment is OK");
    else PUT_LINE ("Mod 4 alignment is incorrect");
    end if;
--
-- Test for mod 8 alignment
--
  if CVT_ADDRESS_TO_INTEGER (REC_4'ADDRESS) mod 8 = 0
    then PUT_LINE ("Mod 8 alignment is OK");
    else PUT_LINE ("Mod 8 alignment is incorrect");
    end if;
```

```
end TST_STATIC_REC_ALIGN;
```

```
--*********************************************************************
-- TST_REC_ALIGN - This procedure tests an Ada implementation's ability
--  to provide for the alignment of stack allocated and heap-allocated
--  records.  The tested alignments are 2, 3, 4, and 8.  The procedure
--  checks the alignment of all compiled records.  Of course, correct
--  alignment could be accidental.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
with SYSTEM;
with UNCHECKED_CONVERSION;
with TEXT_IO;
use TEXT_IO;
procedure TST_REC_ALIGN is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  function CVT_ADDRESS_TO_INTEGER is new UNCHECKED_CONVERSION
     (SOURCE => SYSTEM.ADDRESS, TARGET => INTEGER);
--
-- Test for mod 2 alignment
--
  type REC_1_TYPE is
    record
      B : BOOLEAN;
      A : INTEGER;
    end record;
  for REC_1_TYPE use
    record at mod 2;
    end record;
--
-- Test for mod 3 alignment
--
  type REC_2_TYPE is
    record
      A : BOOLEAN;
      B : INTEGER;
    end record;
--  for REC_2_TYPE use
--    record at mod 3;
--    end record;
--
-- Test for mod 4 alignment
--
  type REC_3_TYPE is
    record
      A : BOOLEAN;
      C : INTEGER;
    end record;
  for REC_3_TYPE use
```

```
    record at mod 4;
    end record;
--
-- Test for mod 8 alignment
--
  type REC_4_TYPE is
    record
      A : BOOLEAN;
      D : INTEGER;
    end record;
  for REC_4_TYPE use
    record at mod 8;
    end record;
--
-- Test for mod 16 alignment
--
  type REC_5_TYPE is
    record
      A : BOOLEAN;
      B : INTEGER;
    end record;
  for REC_5_TYPE use
    record at mod 16;
    end record;
--
-- DUM_1, DUM_2, and DUM_3 are used to force an odd address
--  for records that are supposed to be even aligned.
--
  DUM_1 : CHARACTER;
  REC_1 : REC_1_TYPE;
--  REC_2 : REC_2_TYPE;
  DUM_2 : CHARACTER;
  REC_3 : REC_3_TYPE;
--  DUM_3 : CHARACTER;
--  REC_4 : REC_4_TYPE;

  type PTR_REC_1_TYPE is access REC_1_TYPE;
  type PTR_REC_2_TYPE is access REC_2_TYPE;
  type PTR_REC_3_TYPE is access REC_3_TYPE;
  type PTR_REC_4_TYPE is access REC_4_TYPE;
  type PTR_REC_5_TYPE is access REC_5_TYPE;

  function CVT_PTR_REC_1_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
    (SOURCE => PTR_REC_1_TYPE, TARGET => INTEGER);
  function CVT_PTR_REC_2_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
    (SOURCE => PTR_REC_2_TYPE, TARGET => INTEGER);
  function CVT_PTR_REC_3_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
    (SOURCE => PTR_REC_3_TYPE, TARGET => INTEGER);
  function CVT_PTR_REC_4_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
    (SOURCE => PTR_REC_4_TYPE, TARGET => INTEGER);
  function CVT_PTR_REC_5_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
```

```
    (SOURCE => PTR_REC_5_TYPE, TARGET => INTEGER);

  PTR_REC_1 : PTR_REC_1_TYPE;
  PTR_REC_2 : PTR_REC_2_TYPE;
  PTR_REC_3 : PTR_REC_3_TYPE;
  PTR_REC_4 : PTR_REC_4_TYPE;
  PTR_REC_5 : PTR_REC_5_TYPE;

  begin

  if CVT_ADDRESS_TO_INTEGER (REC_1'ADDRESS) mod 2 = 0
    then PUT_LINE ("Mod 2 alignment is OK");
    else PUT_LINE ("Mod 2 alignment is incorrect");
    end if;

--  if CVT_ADDRESS_TO_INTEGER (REC_2'ADDRESS) mod 3 = 0
--    then PUT_LINE ("Mod 3 alignment is OK");
--    else PUT_LINE ("Mod 3 alignment is incorrect");
--    end if;

  if CVT_ADDRESS_TO_INTEGER (REC_3'ADDRESS) mod 4 = 0
    then PUT_LINE ("Mod 4 alignment is OK");
    else PUT_LINE ("Mod 4 alignment is incorrect");
    end if;

--  if CVT_ADDRESS_TO_INTEGER (REC_4'ADDRESS) mod 8 = 0
--    then PUT_LINE ("Mod 8 alignment is OK");
--    else PUT_LINE ("Mod 8 alignment is incorrect");
--    end if;

  PTR_REC_1 := new REC_1_TYPE;
  if CVT_PTR_REC_1_TYPE_TO_INTEGER (PTR_REC_1) mod 2 = 0
    then PUT_LINE ("Mod 2 alignment of heap objects is OK");
    else PUT_LINE ("Mod 2 alignment of heap objects is incorrect");
    end if;

  PTR_REC_2 := new REC_2_TYPE;
  if CVT_PTR_REC_2_TYPE_TO_INTEGER (PTR_REC_2) mod 3 = 0
    then PUT_LINE ("Mod 3 alignment of heap objects is OK");
    else PUT_LINE ("Mod 3 alignment of heap objects is incorrect");
    end if;

  PTR_REC_3 := new REC_3_TYPE;
  if CVT_PTR_REC_3_TYPE_TO_INTEGER (PTR_REC_3) mod 4 = 0
    then PUT_LINE ("Mod 4 alignment of heap objects is OK");
    else PUT_LINE ("Mod 4 alignment of heap objects is incorrect");
    end if;

  PTR_REC_4 := new REC_4_TYPE;
  if CVT_PTR_REC_4_TYPE_TO_INTEGER (PTR_REC_4) mod 8 = 0
    then PUT_LINE ("Mod 8 alignment of heap objects is OK");
```

```
  else PUT_LINE ("Mod 8 alignment of heap objects is incorrect");
  end if;

PTR_REC_5 := new REC_5_TYPE;
if CVT_PTR_REC_5_TYPE_TO_INTEGER (PTR_REC_5) mod 16 = 0
  then PUT_LINE ("Mod 16 alignment of heap objects is OK");
  else PUT_LINE ("Mod 16 alignment of heap objects is incorrect");
  end if;

end TST_REC_ALIGN;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- STATIC_REC_DATA  - This package provides static record alignment
--   data that is imported by the procedure TST_STATIC_REC_ALIGN.
--   The record types in this package are aligned on mod 2, 3, 4, and
--   8 boundaries.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
package STATIC_REC_DATA is

  type REC_1_TYPE is
    record
      A : INTEGER;
    end record;
  for REC_1_TYPE use
    record at mod 2;
    end record;

--  type REC_2_TYPE is
--    record
--      B : INTEGER;
--    end record;
--  for REC_2_TYPE use
--    record at mod 3;
--    end record;

  type REC_3_TYPE is
    record
      C : INTEGER;
    end record;
  for REC_3_TYPE use
    record at mod 4;
    end record;

  type REC_4_TYPE is
    record
      D : INTEGER;
    end record;
  for REC_4_TYPE use
    record at mod 8;
    end record;
--
-- DUM_1, DUM_2, and DUM_3 are used to force odd addresses for
--   the records whose alignments are supposed to be even.
--
  DUM_1 : CHARACTER;
  REC_1 : REC_1_TYPE;
--  REC_2 : REC_2_TYPE;
  DUM_2 : CHARACTER;
```

```
REC_3 : REC_3_TYPE;
DUM_3 : CHARACTER;
REC_4 : REC_4_TYPE;
end STATIC_REC_DATA;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_BIT_NUM - This procedure determines the bit numbering
--   direction of an Ada implementation.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
with TEXT_IO;
use TEXT_IO;
with UNCHECKED_CONVERSION;
procedure TST_BIT_NUM is
  type TINY_TYPE is range 0. .;
  type SMALL_TYPE is range 0 .. (2 ** 7) - 1;
  type BYTE_TYPE is range 0 .. (2 ** 8) - 1;
  type REC_TYPE is
    record
      TST_BIT : TINY_TYPE;
      DUM_1   : SMALL_TYPE;
    end record;
  for REC_TYPE use
    record
      TST_BIT at 0 range 0..0;
      DUM_1 at 0 range 1..7;
    end record;

  package BYTE_IO is new INTEGER_IO(BYTE_TYPE);
  use BYTE_IO;

  function CVT_REC_TYPE_TO_BYTE_TYPE is new UNCHECKED_CONVERSION
     (SOURCE => REC_TYPE, TARGET => BYTE_TYPE);

  REC  : REC_TYPE;
  INT8 : BYTE_TYPE;

  begin
  REC.TST_BIT := 1;
  REC.DUM_1 := 0;
  INT8 := CVT_REC_TYPE_TO_BYTE_TYPE (REC);
  if INT8 = 1
    then PUT ("This implementation numbers bits right to left");
    else if INT8 = 255
           then PUT ("This implementation numbers bits left to right");
           else
             PUT ("The test failed;  the value of INT8 is:");
             PUT (INT8);  NEW_LINE;
          end if;
  end if;
  end TST_BIT_NUM;
```

```
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_ADDR - This procedure is designed to test an Ada
--  implementation's capabilities for ADDRESS clauses and
--  the ADDRESS attribute.
--
--  Six separate cases are tested:
--   1. An integer in a package (PKG_SPOT) (static)
--   2. An integer in a procedure (PROC_SPOT) (stack allocated)
--   3. An unconstrained array of integers in a procedure (LIST)
--   4. An integer in a task (TSK_SPOT)
--   5. The value of a pointer, PTR_1, is compared with
--      PTR_1.all'ADDRESS for equality.
--   6. The implementation is tested to determine whether
--      multidimensional arrays are stored in row-major or
--      column-major order, using the ADDRESS attribute.
--      (Addresses of array elements could be wrong if the
--       incorrect assumption is made about the order of storage
--       of multidimensional array elements.)
--
--  The four variables, PGK_SPOT, PROC_SPOT, LIST, and TSK_DPOT,
--   are placed at a specific addresses and then the ADDRESS
--   attribute is used to determine their addresses, which are then printed.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
--  The VAX Ada function TO_ADDRESS is used in this procedure to
--   convert constants to ADDRESS type for use in address representation
--   clauses.  The VAX documentation states that universal constants
--   will serve as ADDRESS values, but the compiler rejects them.
--   Maybe my documentation is ahead of my compiler version.
--
with UNCHECKED_CONVERSION;
with TEXT_IO;
use TEXT_IO;
with SYSTEM;
procedure TST_ADDR is
  function CVT_ADDRESS_TO_INTEGER is new UNCHECKED_CONVERSION
      (SOURCE => SYSTEM.ADDRESS, TARGET => INTEGER);
  package TST_PKG is
    procedure PROC (LENGTH : in INTEGER);
    task TSK;
    end TST_PKG;

  package body TST_PKG is
    ADDR        : INTEGER;
    PKG_SPOT    : INTEGER;
    for PKG_SPOT use at SYSTEM.TO_ADDRESS(60);
    type UNCONST_LIST is array (INTEGER range <>) of INTEGER;
    package INT_IO is new INTEGER_IO(INTEGER);
```

```
use INT_IO;

procedure PROC (LENGTH: in INTEGER) is
  PROC_SPOT : INTEGER;
  for PROC_SPOT use at SYSTEM.TO_ADDRESS(72);
  LIST      : UNCONST_LIST (1 .. LENGTH);
  for LIST use at SYSTEM.TO_ADDRESS(84);
  type INT_PTR_TYPE is access INTEGER;
  function CVT_INT_PTR_TYPE_TO_INTEGER is new UNCHECKED_CONVERSION
     (SOURCE => INT_PTR_TYPE, TARGET => INTEGER);
  PTR_1     : INT_PTR_TYPE;
  INT_ADDR_1: INTEGER;
  INT_ADDR_2: INTEGER;
  type MAT_TYPE is array (1..2, 1..2) of INTEGER;
  MAT       : MAT_TYPE;

  begin
  PUT ("MEMORY_SIZE is:");
  PUT (SYSTEM.MEMORY_SIZE);   NEW_LINE;
  PUT ("Address of PKG_SPOT is (Should be 60):");
  PUT (CVT_ADDRESS_TO_INTEGER (PKG_SPOT'ADDRESS));  NEW_LINE;
  PUT ("Address of PROC_SPOT is (Should be 72):");
  PUT (CVT_ADDRESS_TO_INTEGER (PROC_SPOT'ADDRESS));   NEW_LINE;
  PUT ("Address of LIST is (Should be 84):");
  PUT (CVT_ADDRESS_TO_INTEGER (LIST'ADDRESS));  NEW_LINE;

  PTR_1 := new INTEGER;
  PTR_1.all := 42;
  INT_ADDR_1 := CVT_INT_PTR_TYPE_TO_INTEGER (PTR_1);
  INT_ADDR_2 := CVT_ADDRESS_TO_INTEGER (PTR_1.all'ADDRESS);
  if INT_ADDR_1 = INT_ADDR_2
    then PUT ("The values of PTR_1 and PTR_1.all'ADDRESS are equal");
    else PUT ("The values of PTR_1 and PTR_1.all'ADDRESS are unequal");
    end if;
  NEW_LINE;

  if CVT_ADDRESS_TO_INTEGER (MAT(1, 2)'ADDRESS)  <
     CVT_ADDRESS_TO_INTEGER (MAT(2, 1)'ADDRESS)
    then
      PUT ("This implementation stores multidimensional arrays");
      PUT (" in row-major order");
    else
      PUT ("This implementation stores multidimensional arrays");
      PUT (" in column-major order");
    end if;
  NEW_LINE;
  end PROC;

task body TSK is
  TSK_SPOT  : INTEGER;
  for TSK_SPOT use at SYSTEM.TO_ADDRESS(96);
```

```
  begin
  PUT ("Address of TSK_SPOT is (Should be 96):");
  PUT (CVT_ADDRESS_TO_INTEGER (TSK_SPOT'ADDRESS));  NEW_LINE;
  end TSK;

end TST_PKG;

begin
TST_PKG.PROC (10);
end TST_ADDR;
```

```
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_REC_COMP - a procedure to test the capabilities of an
--  Ada system to accept and correctly follow several different
--  specifications of representation clauses for record components.
--
--  The tested  features are:
--   1. Record components that cross storage unit boundaries.
--   2. Placing a default size element on a non-boundary.
--   3. Placing an odd-sized record as a component at a
--      non-boundary in a record.
--   4. Are single-bit components allowed?
--   5. Can a 32-bit integer be placed at a non-byte boundary?
--   6. Can a 32-bit integer be placed at a byte boundary?
--   7. Can a FLOAT (assumed to be 32 bits) be placed at a non-byte
--      boundary?
--   8. Can a FLOAT (assumed to be 32 bits) be placed at a byte
--      boundary?
--   9. Can POSITION be used to compute the bit offset from the
--      beginning of a record to the first bit of a component?
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
with TEXT_IO;
use TEXT_IO;
with SYSTEM;
procedure TST_REC_COMP is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
  type TINY_TYPE is range 0 .. 1;
  type SMALL_TYPE is range 0 .. (2 ** 5) - 1;
  type BYTE_TYPE is range 0 .. (2 ** 8) - 1;
  type MEDIUM_TYPE is range 0 .. (2 ** 9) - 1;
  type BIG_TYPE is range 0 .. (2 ** 18) - 1;
  type HUGE_TYPE is range -(2 ** 31) .. (2 ** 31) - 1;
--
-- Components crossing storage unit boundaries
--
  type REC_1_TYPE is
    record
      A : SMALL_TYPE;
      B : MEDIUM_TYPE;
      C : BIG_TYPE;
    end record;
  for REC_1_TYPE use
    record
      A at 0 range 0 .. 4;
      B at 0 range 5 .. 13;     -- Crosses byte boundary
      C at 0 range 14 .. 31;    -- Crosses word boundary
    end record;
```

```
--
-- Placing a default size element on a non-boundary
--
  type REC_2_TYPE is
    record
      A : SMALL_TYPE;
      B : BOOLEAN;
      C : BIG_TYPE;
    end record;
  for REC_2_TYPE use
    record
      A at 0 range 0 .. 4;
      B at 0 range 5 .. 13;
      C at 0 range 14 .. 31;
    end record;
--
-- REC_3_TYPE is a record type for the following test
--
  type REC_3_TYPE is
    record
      A : SMALL_TYPE;     -- 5 bits
      B : MEDIUM_TYPE;    -- 9 bits
    end record;
  for REC_3_TYPE use
    record
      A at 0 range 0 .. 4;
      B at 0 range 5 .. 13;
    end record;
  for REC_3_TYPE'SIZE use 14;
--
-- Test for placing a record of odd size at an odd spot
--
  type REC_4_TYPE is
    record
      A : SMALL_TYPE;
      X : REC_3_TYPE;
    end record;
  for REC_4_TYPE use
    record
      A at 0 range 0 .. 4;
      X at 0 range 5 .. 19;
    end record;
--
--  Test for use of a single-bit component
--
  type REC_5_TYPE is
    record
      A : SMALL_TYPE;
      B : SMALL_TYPE;
      C : SMALL_TYPE;
      D : TINY_TYPE;
```

```
    end record;
  for REC_5_TYPE use
    record
      A at 0 range 0..4;
      B at 0 range 5..9;
      C at 0 range 10..14;
      D at 0 range 15..15;
    end record;
--
-- Test for placing a 32-bit integer on an odd boundary
--
  type REC_6_TYPE is
    record
      A : SMALL_TYPE;
      B : HUGE_TYPE;
    end record;
  for REC_6_TYPE use
    record
      A at 0 range 0..4;
      B at 0 range 5..36;
    end record;
--
-- Test for placing a 32-bit integer on a byte boundary
--
  type REC_7_TYPE is
    record
      A : BYTE_TYPE;
      B : HUGE_TYPE;
    end record;
  for REC_7_TYPE use
    record
      A at 0 range 0..7;
      B at 1 range 0..31;
    end record;
--
-- Test for placing a FLOAT type at an odd boundary
--
  type REC_8_TYPE is
    record
      A : SMALL_TYPE;
      B : FLOAT;
    end record;
--  for REC_8_TYPE use
--    record
--      A at 0 range 0..4;
--      B at 0 range 5..36;
--    end record;
--
-- Test for placing a FLOAT type at a byte boundary
--
  type REC_9_TYPE is
```

```
    record
      A : BYTE_TYPE;
      B : FLOAT;
    end record;
  for REC_9_TYPE use
    record
      A at 0 range 0..7;
      B at 1 range 0..31;
    end record;

--
-- Test of the POSITION attribute
--
  type REC_10_TYPE is
    record
      A : SMALL_TYPE;
      B : MEDIUM_TYPE;
      C : MEDIUM_TYPE;
      D : SMALL_TYPE;
    end record;
  for REC_10_TYPE use
    record
      A at 0 range 0..4;
      B at 0 range 5..13;
      C at 0 range 14..22;
      D at 0 range 23..27;
    end record;

  REC_1 : REC_1_TYPE;
  REC_2 : REC_2_TYPE;
  REC_3 : REC_3_TYPE;
  REC_4 : REC_4_TYPE;
  REC_5 : REC_5_TYPE;
  REC_6 : REC_6_TYPE;
  REC_7 : REC_7_TYPE;
  REC_8 : REC_8_TYPE;
  REC_9 : REC_9_TYPE;
  REC_10 : REC_10_TYPE;
  BIT_POS: INTEGER;

  begin
  PUT_LINE ("The following output indicates correct results");
  PUT_LINE ("  for an implementation that has storage units");
  PUT_LINE ("  of 8 bits");
  NEW_LINE;
  PUT ("For REC_1, the first bit of B is (Should be 5):");
  PUT (REC_1.B'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_1, the first bit of C is (Should be 6):");
  PUT (REC_1.C'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_2, the first bit of B is (Should be 5):");
  PUT (REC_2.B'FIRST_BIT);   NEW_LINE;
```

```
  PUT ("For REC_2, the first bit of C is (Should be 6):");
  PUT (REC_2.C'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_3, the first bit of B is (Should be 5):");
  PUT (REC_3.B'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_4, the first bit of X is (Should be 5):");
  PUT (REC_4.X'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_4, the last bit of X is (Should be 19):");
  PUT (REC_4.X'LAST_BIT);    NEW_LINE;
  PUT ("For REC_5, the first bit of D is (Should be 7):");
  PUT (REC_5.D'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_5, the last bit of D is (Should be 7):");
  PUT (REC_5.D'LAST_BIT);    NEW_LINE;
  PUT ("For REC_6, the first bit of B is (Should be 5):");
  PUT (REC_6.B'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_7, the first bit of B is (Should be 0):");
  PUT (REC_7.B'FIRST_BIT);   NEW_LINE;
--  PUT ("For REC_8, the first bit of B is (Should be 5):");
--  PUT (REC_8.B'FIRST_BIT);   NEW_LINE;
  PUT ("For REC_9, the first bit of B is (Should be 0):");
  PUT (REC_9.B'FIRST_BIT);   NEW_LINE;
  PUT_LINE ("For REC_10, the bit offset of component D from the ");
  PUT ("beginning of the record (via POSITION) is (Should be 23):");
  BIT_POS := REC_10.D'POSITION * SYSTEM.STORAGE_UNIT
             + REC_10.D'FIRST_BIT;
  PUT (BIT_POS);  NEW_LINE;

  end TST_REC_COMP;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- TST_UNC_CVS - A procedure to test the capabilities of an
--  Ada implementation to support unchecked conversions.
--  Tested features:
--  1. Convert a larger integer to a smaller integer
--  2. Convert a smaller integer to a larger integer
--  3. Convert an integer to an integer of equal size
--
-- NOTE: This procedure produces no output.  Although testing
--       the correctness of the results of the tested conversions,
--       there is no implementation-independent way of doing it.
--
--  Author: R.W. Sebesta
--  Date:   July, 1988
--
with UNCHECKED_CONVERSION;
procedure TST_UNC_CVS is
  type SHORT_INT_TYPE is range 0 .. (2 ** 8) - 1;
  for SHORT_INT_TYPE'SIZE use 8;
  type LONG_INT_1_TYPE is range 0 .. (2 ** 16) - 1;
  for LONG_INT_1_TYPE'SIZE use 16;
  type LONG_INT_2_TYPE is range 0 .. (2 ** 16) - 1;
  for LONG_INT_2_TYPE'SIZE use 16;

  SHORT         : SHORT_INT_TYPE;
  LONG_1        : LONG_INT_1_TYPE;
  LONG_2        : LONG_INT_2_TYPE;

  function CVT_SHORT_INT_TYPE_TO_LONG_INT_1_TYPE is new UNCHECKED_CONVERSION
     (SOURCE => SHORT_INT_TYPE, TARGET => LONG_INT_1_TYPE);
  function CVT_LONG_INT_1_TYPE_TO_SHORT_INT_TYPE is new UNCHECKED_CONVERSION
     (SOURCE =>LONG_INT_1_TYPE, TARGET => SHORT_INT_TYPE);
  function CVT_LONG_INT_1_TYPE_TO_LONG_INT_2_TYPE is new
     UNCHECKED_CONVERSION (SOURCE => LONG_INT_1_TYPE,
       TARGET => LONG_INT_2_TYPE);

  begin
  SHORT := 27;
  LONG_1 := 300;
  LONG_2 := 300;
  LONG_1 := CVT_SHORT_INT_TYPE_TO_LONG_INT_1_TYPE (SHORT);
  SHORT := CVT_LONG_INT_1_TYPE_TO_SHORT_INT_TYPE (LONG_1);
  LONG_2 := CVT_LONG_INT_1_TYPE_TO_LONG_INT_2_TYPE (LONG_1);
  end TST_UNC_CVS;
```

10. Appendix B - Transportability Requirements Report

Performance Dependencies: Check if YES

Direct Execution of Interrupt Entries: ( )

Rendezvous without context switch optimization: ( )

Other Tasking optimization (explain below): ( )

_____

_____

_____

Fast propagation of Exceptions to handler: ( )

Use of Package MACHINE_CODE (LOC: _____): ( )

High Resolution of type DURATION: _____ ( )

High Resolution of type TIME: _____ ( )

Numeric Dependencies:

Specific Rounding Algorithm Required: ( )

Fixed Point Type with 'Small other than a power of two: ( )

Input/Output Dependencies:

Temporary Files: ( )

Shared Files: ( )

FORM Argument String is used: ( )

Buffering Mechanism is Critical: ( )

Length of File Names Required: _____ ( )

Program Initiation

What requirements are placed on the environment task prior to program initiation? For example:

How are parameters for the main subprogram made available?

Are interrupts disabled?

Memory parity has been correctly set?

_____

_____

_____

Other:

BIT ORDERING for Representation Clauses:

BIT numbers are equal to respective power of two: ( )

BIT numbers are non-standard (as follows): ( )

_____

_____

Address Clause Representation:

_____

Non-Ada Code (HOL) is Used (LOC:_____): ( )

Implementation Defined Attributes used: ( )

_____